

1

Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform

	1.1	Introduction.....	1-2
	1.2	Model descriptions	1-5
		PRAM Model • The Work-Depth Methodology • XMT Programming Model • XMT Execution Model • Clarifications of the Modeling	
	1.3	An Example for Using the Methodology: Summation	1-15
	1.4	Empirical Validation of the Performance Model ..	1-16
		Performance Comparison of Implementations • Scaling with the Architecture	
	1.5	Conclusion of Extended Summary	1-19
	1.6	Compiler Optimizations	1-21
		Nested Parallel Sections • Clustering • Prefetching	
	1.7	Prefix-Sums	1-24
		Synchronous Prefix-Sums • No-Busy-Wait Prefix-Sums • Clustering for Prefix-sums • Comparing Prefix-sums Algorithms	
Uzi Vishkin <i>University of Maryland Institute for Advanced Computer Studies and Dept. of Electrical and Computer Engineering, University of Maryland, College Park</i>	1.8	Programming Parallel Breadth-First Search Algorithms	1-28
		Nested Spawn BFS • Flattened BFS • Single-spawn and k -spawn BFS	
George C. Caragea <i>Dept. of Computer Science, University of Maryland, College Park</i>	1.9	Execution of Breadth-First Search Algorithms....	1-30
	1.10	Adaptive Bitonic Sorting	1-33
	1.11	Shared Memory Sample Sort	1-35
Bryant C. Lee <i>Dept. of Computer Science, Carnegie-Mellon University</i>	1.12	Sparse Matrix - Dense Vector Multiplication.....	1-36
	1.13	Speed-ups over Serial execution	1-37
	1.14	Conclusion	1-38

Justin Rattner, CTO, Intel, Electronic News, March 13, 2006: “It is better for Intel to get involved in this now so when we get to the point of having 10s and 100s of cores we will have the answers. There is a lot of architecture work to do to release the potential, and we will not bring these products to market until we have good solutions to the programming problem.” [underline added]

Abstract

Given a PRAM algorithm, the current paper suggests a methodology for converting it into an efficient parallel program for explicit multi-threading (XMT)—a chip-multiprocessor architecture platform under development at the University of Maryland. Given a text on PRAM algorithms as well as our XMT system tool-chain, comprising a compiler and an instance of the hardware, the methodology links the algorithms and the system. Currently available instances of the hardware include a cycle-accurate simulator derived from a synthesizable hardware description, as well as its just completed first commitment to silicon. The latter is based on FPGA technology.

More concretely, a widely used methodology for advancing parallel algorithmic thinking into parallel algorithms is revisited and extended into a methodology for advancing parallel algorithms to XMT (or “PRAM-On-Chip”) programs. A performance cost model for XMT is also presented. It uses as complexity metrics the length of sequence of round trips to memory (LSRTM) and queuing delay (QD) from memory access queues, in addition to standard PRAM computation costs of work and depth. Highlighting the importance of LSRTM in determining performance is another contribution of the paper.

It was unavoidable to have XMT architecture choices impact the XMT performance cost model being proposed. However, the proposed model is quite general, as it abstracts away most low-level details. We believe that the model is quite robust, and that the model and methodology presented are of interest beyond the context of particular XMT architecture choices.

Part I. Extended Summary

1.1 Introduction

Parallel programming is currently a difficult task. But, it does not have to be that way. Current methods tend to be coarse-grained and use either a shared memory or a message passing model. These methods often require the programmer to think in a way that takes into account details of memory layout or architectural implementation, leading the 2003 NSF Blue-Ribbon Advisory Panel on Cyberinfrastructure to opine that: to many users, programming existing parallel computers is still as intimidating and time consuming as programming in assembly language. Consequently, to date the outreach of parallel computing has fallen short of historical expectations. With the ongoing transition to chip multiprocessing (“multi-cores”), industry is considering the parallel programming problem a key bottleneck for progress in the commodity processor space. In recent decades thousands of papers have been written on algorithmic models that accommodate simple representation of concurrency. This effort brought about a fierce debate between a considerable number of schools-of-thoughts, with one of the approaches, the “PRAM approach”, emerging as a clear winner in this “battle of ideas”. Three of the main standard undergraduate computer science texts that came out in 1988-90 [5, 18, 43] chose to include large chapters on PRAM algorithms. The PRAM was the model of choice for parallel algorithms in all major algorithms/theory communities and was taught everywhere. This win did not register in the collective memory as the clear and decisive victory it was since: (i) at about the same time

(early 1990s), it became clear that it will not be possible to build a machine that can look to the performance programmer as a PRAM using 1990s technology, (ii) saying that the PRAM approach can never become useful became a mantra, and PRAM-related work came to a near halt, and (iii) apparently, most people made an extra leap into accepting this mantra.

The Parallel Random Access Model (PRAM) is an easy model for parallel algorithmic thinking and for programming. It abstracts away architecture details by assuming that many memory accesses to a shared memory can be satisfied within the same time as a single access. Having been developed mostly during the 1980s and early 1990s in anticipation of a parallel programmability challenge, PRAM algorithmics provides the second largest algorithmic knowledge base right next to the standard serial knowledge base.

With the continuing increase of silicon capacity, it has become possible to build a single-chip parallel processor. Such demonstration has been the purpose of the Explicit Multi-Threading (XMT) project [60, 47] that seeks to prototype a PRAM-On-Chip vision, as on-chip interconnection networks can provide enough bandwidth for connecting processors to memories [7].

The XMT framework, reviewed briefly in the current paper, provides a quite broad computer system platform. It includes an SPMD (Single Program, Multiple Data) programming model that relaxes the lock-step aspect of the PRAM model, where each parallel step consists of concurrent operations all performed prior to the next parallel step.

The XMT programming model uses thread-level parallelism (TLP), where threads are defined by the programming language and handled by its implementation. The threads tend to be short and are not operating system threads. The overall objective for multi-threading is reducing single-task completion time. While there have been some success stories in compiler effort to automatically extract parallelism from serial code [4, 2], it is mostly agreed that compilers alone are generally insufficient for extracting parallelism from “performance code” written in languages such as C. Henceforth, we assume that the performance programmer is responsible to extract and express the parallelism from the application.

Several multi-chip multiprocessor architectures targeted implementation of PRAM algorithms, or came close to that: (i) The NYU Ultracomputer project viewed the PRAM as providing theoretical yardstick for limits of parallelism as opposed to a practical programming model [50]. (ii) The Tera/Cray Multi-threaded Architecture (MTA) advanced Burton Smith’s 1978 HEP novel hardware design. Some authors have stated that an MTA with large number of processors looks almost like a PRAM; see [15], [6]. (iii) The SB-PRAM may be the first whose declared objective was to provide emulation of the PRAM [39]. A 64-processor prototype has been built [21]. (iv) Although a language rather than an architecture, NESL [10] sought to make PRAM algorithms easier to express using a functional program that is compiled and run on standard multi-chip parallel architectures. However, PRAM theory has generally not reached out beyond academia and it is still undecided whether a PRAM can provide an effective abstraction for a proper design of a multi-chip multi-processor, due to limits on the bandwidth of such an architecture [19]. While we did not find an explicit reference to the PRAM model in MIT-Cilk related papers, the short tutorial [42] presents similar features to how we would have approached a first draft of divide-and-conquer parallel programs. This applies especially to the incorporation of work and depth in such first draft program. As pointed out in Section 1.6.2, we also do not claim in the current paper original contributions beyond the MIT-Cilk on issues concerning clustering, and memory utilization related to implementation of nested spawns.

Guided by the fact that the number of transistors on a chip keeps growing and already exceeds one billion, up from less than 30,000 circa 1980, the main insight behind XMT is

as follows. Billion transistor chips allow the introduction of a high-bandwidth low-overhead on-chip multi-processor. It also allows an evolutionary path from serial computing. The drastic slow down in clock rate improvement for commodity processors since 2003 is forcing vendors to seek single task performance improvements through parallelism. While some have already announced growth plans to 100-core chips by the mid-2010s, they are yet to announce algorithms, programming and machine organization approaches for harnessing these enormous hardware resources toward single task completion time. XMT addresses these issues.

Some key differences between XMT and the above multi-chip approaches are: (i) its larger bandwidth, benefiting from the on-chip environment; (ii) lower latencies to shared memory, since an on-chip approach allows on-chip shared caches; (iii) effective support for serial code; this may be needed for backward compatibility for serial programs, or for serial sections in PRAM-like programs; (iv) effective support for parallel execution where the amount of parallelism is low; certain algorithms (e.g., breadth first-search (BFS) on graphs presented later) have particularly simple parallel algorithms; some are only a minor variation of the serial algorithm; since they may not offer sufficient parallelism for some multi-chip architectures, such important algorithms had no merit for these architectures; and (v) XMT introduced a so-called *Independence of Order Semantics* (IOS), which means that each thread executes at its own pace and any ordering of interactions among threads is valid. If more than one thread may seek to write to the same shared variable this would be in line with the PRAM “arbitrary CRCW” convention (see section 1.2.1). This IOS feature improves performance as it allows processing with whatever data is available at the processing elements and saves power as it reduces synchronization needs. An IOS feature could have been added to multi-chip approaches providing some, but apparently not all the benefits.

While most PRAM-related approaches tended to emphasize competition with (massively parallel) parallel computing approaches, not falling behind modern serial architectures has been an objective for XMT.

XMT can also support standard application programming interfaces (APIs) such as those used for graphics (e.g. OpenGL) or circuit design (e.g. VHDL). For example, [30]: (i) demonstrated speedups exceeding a hundred fold over serial computing for gate-level VHDL simulations implemented on XMT, and (ii) explained how these results can be achieved by automatic compiler extraction of parallelism. Effective implementation of such APIs on XMT would allow an application programmer to take advantage of parallel hardware with few or no changes to an existing API.

Contributions.

The main contributions of this paper are as follows. (1) Presenting a programming methodology for converting PRAM algorithms to PRAM-on-chip programs. (2) Performance models used in developing a PRAM-On-Chip program are introduced, with a particular emphasis on a certain complexity metric, the length of the sequence of round trips to memory (LSRTM). While the PRAM algorithmic theory is pretty advanced, many more practical programming examples need to be developed. For standard serial computing, examples for bridging the gap between algorithm theory and practice of serial programming abound, simply because it has been practiced for so long. See also [53]. A similar knowledge base needs to be developed for parallel computing. (3) The current paper provides a few initial programming examples for the work ahead. (4) Alternatives to the strict PRAM model that by further suppressing of details provide (even) easier-to-think frameworks for parallel algorithms and programming development are also discussed. And last, but not

least, these contributions have a much broader reach than the context in which they are presented.

To improve readability of this long paper the presentation comes in two parts. Part I is an extended summary presenting the main contributions of the paper. Part II supports Part I with explanations and examples making the paper self-contained.

Performance models used in developing a PRAM-On-Chip program are described in section 1.2. An example of using the models is given in section 1.3. Some empirical validation of the models is presented in section 1.4. Section 1.5 concludes Part I – the extended summary of the paper. Part II begins with Section 1.6, where compiler optimizations that could affect the actual execution of programs are discussed. Section 1.7 gives another example for applying the models to the prefix sums problem. Section 1.8 presents Breadth-First Search (BFS) in the PRAM-On-Chip Programming Model. Section 1.9 explains the application of compiler optimizations to BFS and compares performance of several BFS implementations. Section 1.10 discusses the Adaptive Bitonic Sorting algorithm and its implementation while Section 1.11 introduces a variant of Sample Sort that runs on a PRAM-On-Chip. Section 1.12 discusses sparse matrix - dense vector multiplication. The discussion on speedups over serial code from the section on empirical validation of the models was deferred to Section 1.13. A conclusion section is followed by a long appendix with quite a few XMTC code examples in support of the text.

1.2 Model descriptions

Given a problem, a “recipe” for developing an efficient XMT program from concept to implementation is proposed. In particular, the stages through which such development needs to pass are presented.

Figure 1.1 depicts the proposed methodology. For context, the figure also depicts the widely used Work-Depth methodology for advancing from concept to a PRAM algorithm, namely, the sequence of models $1 \rightarrow 2 \rightarrow 3$ in the figure. For developing a XMT implementation, we propose following the sequence of models $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, as follows. Given a specific problem, an algorithm design stage will produce a High-Level description of the parallel algorithm, in the form of a sequence of steps, each comprising a *set* of concurrent operations (box 1). In a first draft, the set of concurrent operations can be implicitly defined. See the BFS example in Section 1.2.2. This first draft is refined to a sequence of steps each comprising now an *ordered sequence* of concurrent operations (box 2). Next, the programming effort amounts to translating this description into a single-program multiple-data (SPMD) program using a high-level XMT programming language (box 4). From this SPMD program, a compiler will transform and reorganize the code to achieve the best performance in the target XMT execution model (box 5). As an XMT programmer gains experience, he/she will be able to skip box 2 (the Work-Depth model) and directly advance from box 1 (high-Level Work-Depth description) to box 4 (high-level XMT program). We also demonstrate some instances where it may be advantageous to skip box 2 because of some features of the programming model (such as some ability to handle nesting of parallelism). In Figure 1.1 this shortcut is depicted by the arrow $1 \rightarrow 4$. Much of the current paper is devoted to presenting the methodology and demonstrating it. We start with elaborating on each model.

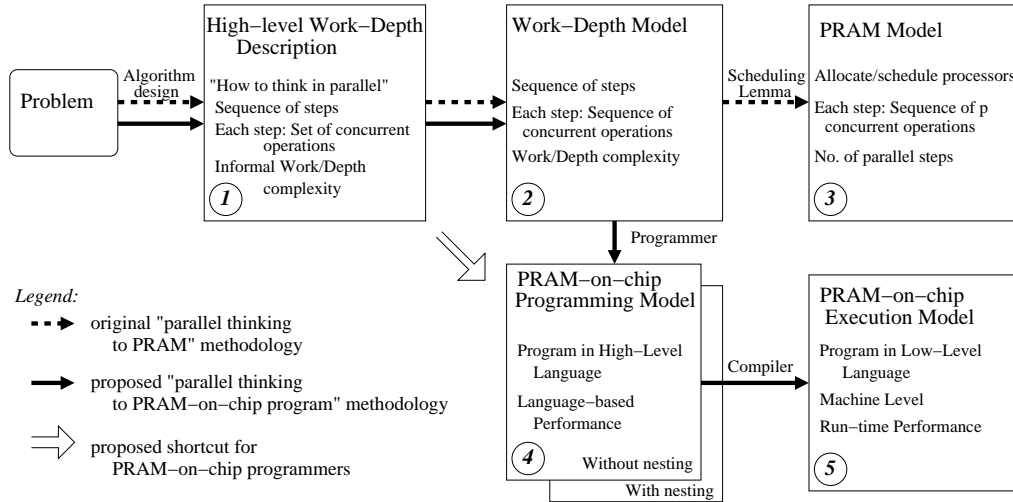


FIGURE 1.1: Proposed Methodology for Developing XMT Programs in view of the Work-Depth Paradigm for Developing PRAM algorithms.

1.2.1 PRAM Model

PRAM (for Parallel Random Access Machine, or Model) consists of p synchronous processors communicating through a global shared memory accessible in unit time from each of the processors. Different conventions exist regarding concurrent access to the memory. For brevity, we only mention arbitrary concurrent-read concurrent-write (CRCW) where simultaneous access to the same memory location for reads or writes are permitted, and concurrent writes to the same memory location result in an arbitrary one among the values sought to be written, but it is not known in advance which one.

There are quite a few sources for PRAM algorithms including [37, 38, 23, 59]. An algorithm in the PRAM model is described as a sequence of parallel time units, or rounds; each round consists of exactly p instructions to be performed concurrently, one per each processor. While the PRAM model is simpler than most parallel models, producing such a description imposes a significant burden on the algorithm designer. Luckily this burden can be somewhat mitigated using the Work-Depth methodology, presented next.

1.2.2 The Work-Depth Methodology

The Work-Depth methodology for designing PRAM algorithms, introduced in [52], has been quite useful for describing parallel algorithms and reasoning about their performance. For example, it was used as the description framework in [37]. The methodology guides algorithm designers to optimize two quantities in a parallel algorithm: *depth* and *work*. Depth represents the number of steps the algorithm would take if unlimited parallel hardware was available, while work is the total number of operations performed, over all parallel steps.

The methodology comprises of two steps: (i) first, produce an informal description of the algorithm in a high-level work-depth model (HLWD), and (ii) refine this description into a fuller presentation in a model of computation called Work-Depth. These two models are described next.

High-Level Work-Depth Description

A HLWD description consists of a succession of parallel rounds, each round being a *set* of any number of instructions to be performed concurrently. Descriptions can come in several flavors, and even implicit descriptions, where the number of instructions is not obvious, are acceptable.

Example: Input: An undirected graph $G(V, E)$ and a source node $s \in V$; the length of every edge in E is 1. Find the length of the shortest paths from s to every node in V . An informal work-depth description of a parallel *breadth-first search* (BFS) algorithm can look as follows. Suppose that the set of vertices V is partitioned into layers. Layer L_i includes all vertices of V whose shortest path from s have i edges. The algorithm works in iterations. In iteration i , layer L_i is found. Iteration 0: node s forms layer L_0 . Iteration i , $i > 0$: Assume inductively that layer L_{i-1} was found. In parallel, consider all the edges (u, v) that have an endpoint u in layer L_{i-1} ; if v is not in a layer L_j , $j < i$, it must be in layer L_i . As more than one edge may lead from a vertex in layer L_{i-1} to v , vertex v is marked as belonging to layer L_i based on one of these edges using the arbitrary concurrent write convention. This ends an informal, high-level work-depth verbal description.

A pseudocode description of an iteration of this algorithm could look as follows:

```
for all vertices v in L(i) pardo
  for all edges e=(v,w) pardo
    if w unvisited
      mark w as part of L(i+1)
```

The above HLWD descriptions challenge us to find an efficient PRAM implementation for an iteration. In particular, given a p -processor PRAM how to allocate processors to tasks to finish all operations of an iterations as quickly as possible? A more detailed description in the Work-Depth model would address these issues.

Work-Depth Model

In the Work-Depth model an algorithm is described in terms of successive time steps, where the concurrent operations in a time step form a sequence; each element in the sequence is indexed from 1 to the number of operations in the step. The Work-Depth model is formally equivalent to the PRAM. For example, a work-depth algorithm with $T(n)$ depth (or time) and $W(n)$ work runs on a p processor PRAM in at most $T(n) + \lfloor \frac{W(n)}{p} \rfloor$ time steps. The simple equivalence proof follows Brent's scheduling principle, which was introduced in [13] for a model of parallel model of computation that was much more abstract than the PRAM (counting arithmetic operations, but suppressing anything else).

Example (continued): We only note here the challenge for coming up with a Work-Depth description for the BFS algorithm: to find a way for listing in a single sequence all the edges whose endpoint is a vertex at layer L_i . In other words, the Work-Depth model does not allow us to leave nesting of parallelism, such as in the pseudocode description of BFS above, unresolved. On the other hand XMT programming should allow nesting of parallel structures, since such nesting provides an easy way for parallel programming. It is also important to note that the XMT architecture includes some limited support for nesting of parallelism: a nested spawn can only spawn k extra threads, where k is a small integer (e.g., $k = 1, 2, 4$ or 8); nested spawn commands are henceforth called either k -spawn, or sspawn (for single spawn). The way in which we suggest to resolve this problem is as follows. The *ideal* long term solution is: (a) allow the programmer free unlimited use of nesting, (b) have it implemented as efficiently as possible by compiler, and (c) make the programmer (especially the "performance programmer") aware of the added cost of using

nesting. However, since our compiler is not yet mature enough to handle this matter, our *tentative* short term solution is presented in Section 1.8, which shows how to build on the support for nesting provided by the architecture. There is merit to this “manual solution” beyond its tentative role until the compiler matures. Such solution should still need to be understood (even after the ideal compiler solution is in place) by performance programmers, so that the impact of nesting on performance is clear to them.

The reason for bringing this issue up this early in the discussion is that it actually demonstrates that our methodology can sometimes proceed directly to the PRAM-like programming methodology, rather than make a “stop” at the Work-Depth model.

1.2.3 XMT Programming Model

A framework for a high-level programming language, the XMT programming model seeks to mitigate two goals: (i) *Programmability*: given an algorithm in the HLWD or Work-Depth models, the programmer’s effort in producing a program should be minimized; and (ii) *Implementability*: effective compiler translation of the program into the XMT execution model should be feasible.

The XMT programming model is fine-grained and SPMD type. Execution can frequently alternate between serial and parallel execution modes. A Spawn command prompts a switch from serial mode to parallel mode (see Figure 1.2). The Spawn command can specify any number of threads. Ideally, each such thread can proceed until termination (a Join command) without ever having to busy-wait or synchronize with other threads. To facilitate that, an *independence of order semantics (IOS)* was introduced. Inspired by the arbitrary concurrent-write convention of the PRAM model, commands such as “prefix-sum” permit threads to proceed even if they try to write into the same memory location.

Some primitives in the XMT programming model follow:

Spawn Instruction. Starts a parallel section. Accepts as parameter the number of parallel threads to start.

Thread-id. A reserved parameter inside a parallel section. Evaluates to the unique thread index. This allows SPMD style programming.

Prefix-sum Instruction. The prefix-sum instruction defines an atomic operation. First assume a global variable B , called base, and a local variable R , called increment, the result of a prefix-sum is: (i) B gets the value $B + R$, and (ii) R gets the original value of B .

Example: Suppose that threads 2, 3 and 5, respectively, execute concurrently the commands $ps(B, R_2)$, $ps(B, R_3)$ and $ps(B, R_5)$, respectively all relating to the same base B and the original values are $B = 0, R_2 = R_3 = R_5 = 1$. IOS allows any order of execution among the 3 prefix-sums commandsnamely, any of the 6 possible permutations. The result of all 6 permutations is $B = 3$. If thread 5 precedes thread 2 that precedes thread 2, we will get $R_5 = 0, R_2 = 1$ and $R_3 = 2$, and if the thread order is 2, 3 and 5 then $R_2 = 0, R_3 = 1, R_5 = 2$. Two example for the use of the prefix sums command are noted next. (i) In the array compaction code in Table 1.1.a, a code example where the prefix-sum command is used is demonstrated. This code example is referenced later in the text. (ii) In order to implement the PRAM arbitrary concurrent write convention, the programmer is guided to do the following: Each location that might be written by several threads has an auxiliary “gatekeeper” location associated with it, initialized with a known value (say 0). When a thread wants to write to the shared location, it first executes a Prefix-sum instruction (e.g., with an increment

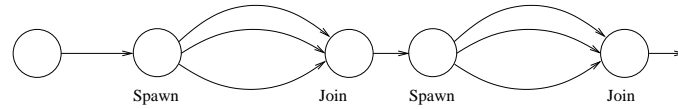


FIGURE 1.2: Switching between serial and parallel execution modes in the XMT programming model. Each parallel thread executes at its own speed, without ever needing to synchronize with another thread

of 1) on the gatekeeper location. Only one thread gets 0 as its result; this thread is allowed to write to the shared location, while the other threads advance to their next instruction without writing.

While, the basic definition of prefix-sum follows the fetch-and-add of the NYU-Ultracomputer [28], XMT uses a fast parallel hardware implementation if R is from a small range (e.g., one bit) and B can fit one of a small number of global registers[57]; otherwise, prefix-sums are obtained using a prefix-sum-to-memory instruction; in the latter case, prefix-sum implementation involves queuing in memory.

Nested parallelism. A parallel thread can be programmed to initiate more threads. However, as noted in Section 1.2.2, this comes with some (tentative) restrictions and cost caveats, due to compiler and hardware support issues. As illustrated with the Breadth-First search example, nesting of parallelism could improve the programmer’s ability to describe algorithms in a clear and concise way.

Note that Figure 1.1 depicts two alternative XMT programming models: without nesting and with nesting. The Work-Depth model maps directly into the programming model without nesting. Allowing nesting could make it easier to turn a description in the High-Level Work-Depth model into a program.

We call the resulting programming language XMTC. XMTC is a superset of the language C which includes statements implementing the above primitives.

Examples of XMTC code

Table 1.1 provides 5 XMTC programs, excluding constructs such as variable and function declarations that were derived from PRAM algorithms. The table demonstrates compactness of code, as may be appreciated by readers familiar with other parallel programming frameworks.

In Table 1.1.a, the language features of XMTC are demonstrated using the *array compaction problem*: given an array of integers $T[0 \dots n - 1]$, copy all its non-zero elements into another array S ; any order will do. The special variable $\$$ denotes the thread-id. Command `spawn(0, n-1)` spawns n threads whose id’s are the integers in the range $0 \dots n - 1$. The `ps(increment, length)` statement executes an atomic prefix-sum command using `length` as the base and `increment` as the increment value. The variable `increment` is local to a thread while `length` is a shared variable which holds the number of non-zero elements copied at the end of the spawn block. Variables declared inside a `spawn` block are local to a thread, and are typically much faster to access than shared memory. Note that on XMT, local thread variables are typically stored into local registers of the executing hardware thread control unit (TCU). The programmer is encouraged to use local variables to store frequently used values. Often, this type of optimizations can also be performed by an optimizing compiler.

<pre> (a) Array compaction /* Input: N numbers in T[0..N-1] * * Output: S contains the non-zero values from T */ length = 0; // shared by all threads spawn(0,n-1) { // start one thread per array element int increment = 1; // local to each thread if(T[\$] != 0) { ps(increment, length); //execute prefix-sum to allocate one entry in S S[increment] = T[\$]; } } </pre>
<pre> (b) k-ary Tree Summation /* Input: N numbers in the leaves of a k-ary tree in a 1D array representation* * Output: The sum of the numbers in sum[0] */ level = 0; while(level < log_k(N)) { // process levels of tree from leaves to root level++; // also compute current_level_start and end_index */ spawn(current_level_start_index, current_level_end_index) { int count, local_sum=0; for(count = 0; count < k; count++) temp_sum += sum[k * \$ + count + 1]; sum[\$] = local_sum; } } </pre>
<pre> (c) k-ary Tree Prefix-Sums /* Input: N numbers in sum[0..N-1] * * Output: the prefix-sums of the numbers in * * prefix_sum[offset_to_1st_leaf..offset_to_1st_leaf+N-1] * * The prefix_sum array is a 1D complete tree representation (See Summation) */ kary_tree_summation(sum); // run k-ary tree summation algorithm prefix_sum[0] = 0; level = log_k(N); while(level > 0) { // all levels from root to leaves spawn(current_level_start_index, current_level_end_index) { int count, local_ps = prefix_sum[\$]; for(count = 0; count < k; count++) { prefix_sum[k*\$ + count + 1] = local_ps; local_ps += sum[k*\$ + count + 1]; } level--; // also compute current_level_start and end_index */ } } </pre>
<pre> (d) Breadth-First Search /* Input: Graph G=(E,V) using adjacency lists (See Programming BFS section) * * Output: distance[N] - distance from start vertex for each vertex * * Uses: level[L][N] - sets of vertices at each BFS level. */ //run prefix sums on degrees to determine position of start edge for each vertex start_edge = kary_prefix_sums(degrees); level[0]=start_node; i=0; while (level[i] not empty) { spawn(0,level_size[i] - 1) { // start one thread for each vertex in level[i] v = level[i][\$]; // read one vertex spawn(0,degree[v]-1) { // start one thread for each edge of each vertex int w = edges[start_edge[v]+\$][2]; // read one edge (v,w) psm(gatekeeper[w],1); //check the gatekeeper of the end-vertex w if gakeeper[w] was 0 { psm(level_size[i+1],1); //allocate one entry in level[i+1] store w in level[i+1]; } } // join } i++; } </pre>
<pre> (e) Sparse Matrix - Dense Vector Multiplication /* Input: Vector b[n], sparse matrix A[m][n] given in Compact Sparse Row form * * (See Sparse Matrix - Dense Vector Multiplication section) * * Output: Vector c[m] = A*b */ spawn(0,m) { // start one thread for each row in A int row_start=row[\$], elements_on_row = row[\$+1]-row_start; spawn(0,elements_on_row-1) { //start one thread for each non-zero element on row // compute A[i][j] * b[j] for all non-zero elements on current row tmpsum[\$]=values[row_start+\$]*b[columns[row_start+\$]]; } c[\$] = kary_tree_summation(tmpsum[0..elts_on_row-1]); // sum up } </pre>

TABLE 1.1 Implementation of some PRAM algorithms in the XMT PRAM-on-chip framework to demonstrate compactness.

To evaluate performance in this model, a *Language-Based Performance Model* is used: performance costs are assigned to each primitive instruction in the language with some accounting rules added for the performance cost of computer programs (e.g., depending on execution sequences). Such performance modeling was used by Aho and Ullman [1] and was generalized for parallelism by Blelloch [10]. The paper [20] used language-based modeling for studying parallel list ranking relative to an earlier performance model for XMT.

1.2.4 XMT Execution Model

The execution model depends on XMT architecture choices. However, as can be seen from the modeling itself, this dependence is rather minimal and should not compromise the generality of the model for other future chip-multiprocessing architectures whose general features are similar. We only review the architecture below and refer to [47] for a fuller description. Class presentation of the overall methodology proposed in the current paper usually breaks here to review, based on [47], the way in which: (i) the XMT apparatus of the program counters and stored program extends the (well-known) von-Neumann serial apparatus, (ii) the switch from serial to parallel mode and back to serial mode is implemented, (iii) virtual threads coming from an XMTC program are allocated dynamically at run time, for load balancing, to thread control units (TCUs), (iv) hardware implementation of prefix-sum enhances the computation, and (v) independence of order semantics (IOS), in particular, and the overall design principle of no-busy-wait finite-state-machines (NBW FSM), in general, allow making as much progress as possible with whatever data and instructions are available.

Possible architecture choices.

A bird eye's view of a slightly more recent version of XMT is presented in Figure 1.3. A number of (say 1024) TCUs are grouped into (say 64) clusters. Clusters are connected to the memory subsystem by a high-throughput, low-latency interconnection network, see e.g., [7]; they also interface with specialized units such as prefix-sum unit and global registers. A hash function is applied to memory addresses in order to provide better load balancing at the shared memory modules. An important component of a cluster is the read-only cache; this is used to store values read from memory by all TCUs in the cluster. In addition, TCUs have prefetch buffers to hold values read by special prefetch instructions. The memory system consists of memory modules each having several levels of cache memories. In general each logical memory address can reside in only one memory module, alleviating cache coherence problems. This connects to including only read-only caches at the clusters. The Master TCU runs serial code, or the serial mode for XMT and is the only TCU that has a local writeable cache. When it hits a Spawn command it initiates a parallel mode by broadcasting the same SPMD parallel code segment to all the TCUs. As each TCU captures its copy, it executes it based on a thread-id assigned to it. A separate distributed hardware system, reported in [47] but not shown in Figure 1.3, ensures that all the thread id's mandated by the current Spawn command are allocated to the TCUs. A sufficient part of this allocation is done dynamically to ensure that no TCU needs to execute more than one thread id, once another TCU is already idle.

Possible role of compiler.

To take advantage of features of the architecture, an XMTC program needs to be translated by an optimizing compiler. In the Execution model, a program could include: (i) Prefetch instructions to bring data from the lower memory hierarch levels either into the

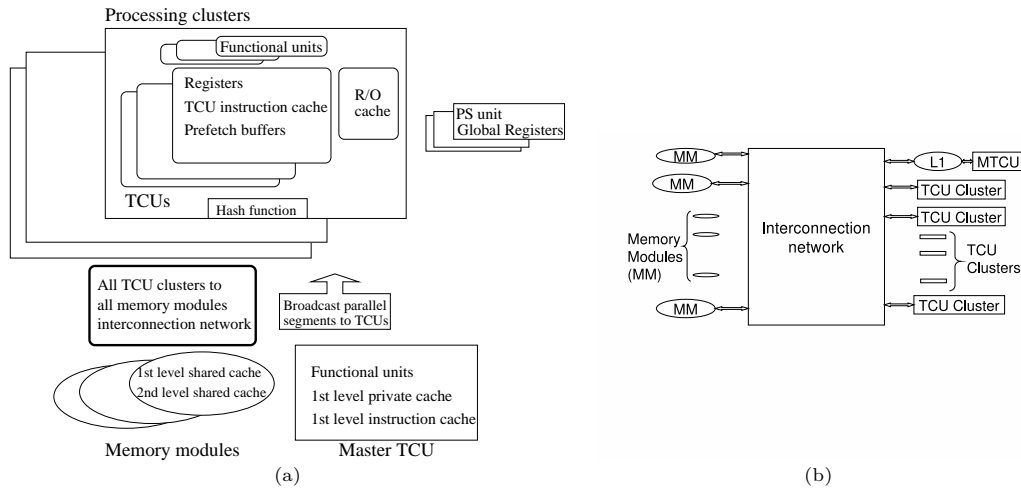


FIGURE 1.3: (a) An overview of the XMT Architecture. (b) Block diagram of memories, TCUs and interconnection network.

shared caches or into the prefetch buffers located at the TCUs; (ii) Broadcast instructions, where some values needed by all, or nearly all TCUs, are broadcasted to all; (iii) Thread clustering: combining shorter virtual threads into a longer thread; and (iv) If the programming model allows nested parallelism, the compiler will use the mechanisms supported by the architecture to implement or emulate it. Compiler optimizations and issues such as nesting and thread clustering are discussed in section 1.6.

Performance Modeling.

The performance modeling of a program first extends the known PRAM notions of *work* and *depth*. Later, a formula for estimating execution time based on these extensions is provided.

The depth of an application in the XMT Execution model must include the following three quantities: (i) *Computation Depth*, given by the number of operations that have to be performed sequentially, either by a thread or while in serial mode. (ii) *Length of Sequence of Round-Trips to Memory (or LSRTM)* which represents the number of cycles on the critical path spent by execution units waiting for data from memory. For example, a read request or prefix-sum instruction from a TCU usually causes a round-trip to memory (or RTM); memory writes in general proceed without acknowledgment, thus not being counted as round-trips, but ending a parallel section implies one RTM used to flush all the data still in the interconnection network to the memory. (iii) *Queuing delay (or QD)* which is caused by concurrent requests to the same memory location; the response time is proportional to the size of the queue.

A prefix-sum (**ps**) statement is supported by a special hardware unit that combines calls from multiple threads into a single multi-operand prefix-sum operation. Therefore, a **ps** statement to the same base over several concurrent threads causes one roundtrip through

the interconnection network to the global register file (1 RTM) and 0 queuing delay, in each of the threads.

The syntax of a prefix-sum to memory (**psm**) statement is similar to **ps** except the base variable is a memory location instead of a global register. As updates to the memory location are queued, the **psm** statement costs 1 RTM and additionally has a queuing delay (QD) reflecting the plurality of concurrent threads executing **psm** with respect to the same memory location.

We can now define the XMT “execution depth” and “execution time”. XMT Execution Depth represents the time spent on the “critical path” (that is, the time assuming unlimited amount of hardware) and is the sum of the computation depth, LSRTM, and QD on the critical path. Assuming that a round-trip to memory takes \mathcal{R} cycles:

$$Execution\ Depth = Computation\ Depth + LSRTM \times \mathcal{R} + QD \quad (1.1)$$

Sometimes, a step in the application contains more *Work* (the total number of instructions executed) to be executed in parallel than what the hardware can handle concurrently. For the additional time spent executing operations outside the critical path (i.e., beyond the Execution Depth), the work of each parallel section needs to be considered separately. Suppose that one such parallel section could employ in parallel up to p_i TCUs. Let $Work_i = p_i * ComputationDepth_i$ be the total computation work of parallel section i . If our architecture has p TCUs and $p_i < p$, we will be able to use only p_i of them, while if $p_i \geq p$, only p TCUs can be used to start the threads, and the remaining $p_i - p$ threads will be allocated to TCUs as they become available; each concurrent allocation of p threads to p TCUs is charged as one RTM to the Execution Time, as per equation 1.2. The total time spent executing instructions outside the critical path over all parallel sections is given in equation 1.3.

$$Thread\ Start\ Overhead_i = \left\lceil \frac{p_i - p}{p} \right\rceil \times \mathcal{R} \quad (1.2)$$

$$Time\ Additional\ Work = \sum_{spawn\ block\ i} \left(\frac{Work_i}{\min(p, p_i)} + ThreadStartOverhead_i \right) \quad (1.3)$$

In the last equation we do not subtract the quantity that is already counted as part of the Execution Depth. The reason is that our objective in the current paper is limited to extending the work-depth upper bound $T(n) + \lfloor \frac{W(n)}{p} \rfloor$, and such double count is possible in that original upper bound as well. Adding up, the execution time of the entire program is:

$$Execution\ Time = Execution\ Depth + Time\ Additional\ Work \quad (1.4)$$

1.2.5 Clarifications of the Modeling

The current paper provides performance modeling that allows weighing alternative implementations of the same algorithm where asymptotic analysis alone is insufficient. Namely, a more refined measure than the asymptotic number of parallel memory accesses was needed.

Next, we point out a somewhat subtle point. Following the path from the HLWD model to the XMT models in Figure 1.1 may be important for optimizing performance, and not only for the purpose of developing a XMT program. Bandwidth is not accounted for in the XMT performance modeling, since the on-chip XMT architecture should be able to provide sufficient bandwidth for an efficient algorithm in the Work-Depth model. The only way in which our modeling accounts for bandwidth is indirect: by first screening an algorithm

through the Work-Depth performance modeling, where we account for work. Now, consider what could have happened had XMT performance modeling not been coupled with Work-Time performance modeling. The program could include excessive speculative prefetching to supposedly improve performance (reduce LSRTM). The subtle point is that the extra prefetches add to the overall work count. Accounting for them in the Work-Depth model prevents this “loophole”.

It is also important to recognize that the model abstracts away some significant details. The XMT hardware has a limited number of memory modules. If multiple requests attempt to access the same module, queuing will occur. While accounting for queuing to the same memory location, the model does not account for queuing accesses to different locations in the same module. Note that hashing memory addresses among modules lessens problems that would occur for accesses with high spatial locality and generally mitigates this type of “hot spots”. If functional units within a cluster are shared between the TCUs, threads can be delayed while waiting for functional units to become available. The model does also not account for these delays.

Our modeling is a first-order approximation of run time. Such analytic results are not a substitute for experimental results, since the latter will not be subject to the approximations described above. In fact, some experimental results as well as a comparison between modeling and simulations are discussed in section 1.4.

Similar to some serial performance modeling, the above modeling assumes that data is found in the (shared) caches. This allows proper comparison to serial computing where data is found in the cache, as the number of clocks to reach the cache for XMT is assumed to be significantly higher than in serial computing; for example, our prototype XMT architecture suggests values that range between 6 and 24 cycles for a round-trip to the first level of cache, depending on the characteristics of the interconnection network and its load level; we took the conservative approach to use the value $\mathcal{R} = 24$ cycles for one RTM for the rest of this paper. We expect that the number of clocks to access main memory should be similar to serial computing and that both for serial computing and for XMT large caches will be built. However, this modeling is inappropriate if XMT is to be compared to Cray MTA where no shared caches are used: for the MTA the number of clocks to access main memory is important and for a true comparison this figure for cache misses on XMT would have to be included as well.

As pointed out earlier, some of the computation work is counted twice in our Execution Time, once as part of the critical path under Execution Depth and once in the Additional Work factor. Future work by us, or others, could refine our analysis into a more accurate model, but with much more involved formulae. In the current paper, we made the choice to stop at this level of detail allowing a concise presentation while still providing relevant results.

Most other researchers that worked on performance modeling of parallel algorithms were concerned with other platforms. They focused on different factors than us. Helman and JáJá [32] measured the complexity of algorithms running on SMPs using the triplet of maximum number of non-contiguous accesses by any processor to main memory, number of barrier synchronizations, and local computation cost. These quantities are less important in a PRAM-like environment. Bader, Cong, and Feo [6] found that in some experiments on the Cray MTA, the costs of non-contiguous memory access and barrier synchronization were reduced almost to zero by multithreading and that performance was best modeled by computation alone. For the latest generation of the MTA architecture, a calculator for performance that includes the parameters of count of trips to memory, number of instructions, and number of accesses to local memory [24] was developed. The RTMs that we count are round trips to the shared cache, which is quite different, as well as queuing at the shared

cache. Another significant difference is that we consider the effect of optimizations such as prefetch and thread clustering. Nevertheless, the calculator should provide an interesting basis for comparison between performance of applications on MTA and XMT. The incorporation of queuing follows the well-known QRQW PRAM model of Gibbons, Matias and Ramachandran [27]. A succinct way to summarize the modeling contribution of the current paper is that unlike previous practice QRQW becomes secondary, though still quite important, to LSRTM.

1.3 An Example for Using the Methodology: Summation

Consider the problem of computing in parallel the sum of N values stored in array A . A High-Level Work-Depth description of the algorithm is as follows: in parallel add groups of k values; apply the algorithm recursively on the $\lceil N/k \rceil$ partial sums until the total sum is computed. This is equivalent to climbing (from leaves towards root) a balanced k -ary tree. An iterative description of this algorithm that fits the Work-Depth model can be easily derived from this. The parameter k is a function of the architecture parameters and the problem size N and is chosen to minimize the estimated running time.

An implementation of this algorithm in the XMT Programming Model is presented in table 1.1.b. Note that a uni-dimensional array is used to store the complete k -ary tree, where the root is stored at element 0, followed by the k elements of the second level from left to right, then the k^2 elements of the second level and so on.

For each iteration of the algorithm, the k partial sums from a node's children have to be read and summed. Prefetching can be used to pipeline the memory accesses for this operation, thus requiring only one round-trip to memory (RTM). An additional RTM is needed to flush all the data to memory at the end of each step. There are no concurrent accesses to the same memory location in this algorithm, thus the queuing delay (QD) is zero. By accounting for the constant factors in our own XMTC implementation, we determined the Computation Depth to be $(3k + 9) \log_k N + 2k + 33$, given that $\log_k N$ iteration are needed. To compute the additional time spent executing outside the critical path (in saturated regime), we determined the computation per tree node to be $C = 3k + 2$ and the total number of nodes processed under this regime to be $Nodes_{sat}$ as in Figure 1.4. An additional step is required to copy the data into the tree's leaves at the start of the execution.

This determines the Execution Work, Additional Work and the Thread Start Overhead terms of the XMT execution time:

$$Execution\ Depth = (2 \log_k N + 1) \times \mathcal{R} + (3k + 9) \log_k N + 2k + 33 \quad (1.5)$$

$$Additional\ Work = \frac{2N + (3k + 2)Nodes_{sat}}{p} + (3k + 2) \log_k p + \left[\frac{Nodes_{sat}}{p} - \log_k \frac{N}{p} \right] \times \mathcal{R} \quad (1.6)$$

To avoid starting too many short threads, an optimization called thread clustering can be applied either by an optimizing compiler or a performance programmer: let c be a constant; start c threads that each run a serial summation algorithm on a contiguous sub-array of N/c values from the input array. Each thread writes its computed partial sum into an array B . To compute the total sum, run the parallel summation algorithm described above on the array B .

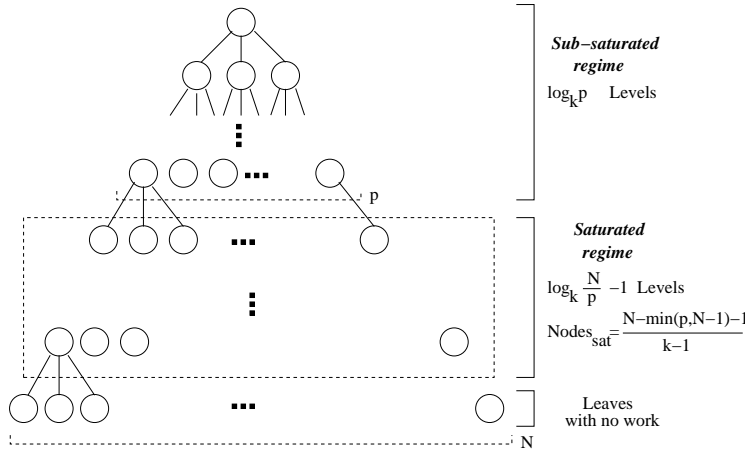


FIGURE 1.4: The $\log_k p$ levels of the tree closest to the root are processed in sub-saturated regime, i.e. there is not enough parallelism in the application to run all the TCUs. The next $\log(N/p) - 1$ levels have more parallelism than the hardware can execute in parallel and some TCUs will run more than one thread (saturated regime). The computation starts at the first level above the leaves.

We now consider how clustering changes the execution time. $SerSum(N)$ and $ParSum(N)$ denote the execution time for serial and the parallel summation algorithms over N values respectively. The serial algorithm loops over N elements and, by using prefetching to always have the next value available before it is needed, it has a running time of $SerSum(N) = 2N + 1 \times \mathcal{R}$.

The first part of the algorithm uses a total of $N - c$ additions evenly divided among p processors, while the second requires the parallel summation to be applied on an input of size c . This gives an execution time for the clustered algorithm of:

$$\text{Execution Time} = SerSum\left(\frac{N - c}{p}\right) + ParSum(c) \quad (1.7)$$

The value of c , where $p \leq c \leq N$, that minimizes the execution time determines the best crossover point for clustering. Suppose $p = 1024$. To allow numerical comparison, we need to assign a value to \mathcal{R} , the number of cycles in one RTM. As noted in section 1.2.5, for the prototype XMT architecture this value is upper bounded by 24 cycles under the assumption that the data is already in the on-chip cache and there is no queuing in the interconnection network or memory.

Since all the clustered threads are equal in length and in the XMT Execution Model they run at the same speed, we found, that when $N \gg p$, the optimal value for c is 1024.

The optimum value for k can be determined by minimizing execution time for a fixed N . For the interesting case when $N \geq p$ (where $p = 1024$), the parallel summation algorithm is only run on $c = 1024$ elements and in this case we found (see Figure 1.6.a) that $k = 8$ is optimal.

1.4 Empirical Validation of the Performance Model

Some empirical validation of the analytic performance model of the previous sections is presented in this section. Given an XMTC program, estimated run-times using the analytic

model are compared against simulated run-times using our XMT simulator. Derived from a synthesizable gate-level description of the XMT architecture, the XMT simulation engine aims at being cycle-accurate. A typical configuration includes 1024 thread control units (TCUs) grouped in 64 clusters and one Master TCU.

A gap between the simulations and the analytic model is to be expected. Not only that the analytic model as presented makes some simplifying assumptions, such as counting each XMTC statement as one cycle, and ignoring contention at the functional units, it provides the same (worst-case) run-time estimates for different input data as long as the input has the same size. On the other hand, at the present time the XMTC compiler and the XMT cycle-accurate simulator lack a number of features that were assumed for the analytic performance model. More specifically, there is no compiler support for prefetching and thread clustering, and only limited broadcasting capabilities are included. (Also, a sleep-wait mechanism noted in Section 1.6.1 is not fully implemented in the simulator, making the single-spawning mechanism less efficient.) All these factors could cause the simulated run-times to be higher than the ones computed by the analytic approach. For that reason, we limit our focus to just studying the relative performance of two or more implementations that solve the same problem. This will enable evaluating programming implementation approaches against the relative performance gain – one of the goals of the full paper.

More information about the XMTC compiler and the XMT cycle-accurate simulator can be found in [8]. An updated version of that document which reflects the most recent version of the XMT compiler and simulator is available from <http://www.umiacs.umd.edu/users/vishkin/XMT/XMTCManual.pdf> and <http://www.umiacs.umd.edu/users/vishkin/XMT/XMTCTutorial.pdf>.

We proceed to describe our experimental results.

1.4.1 Performance Comparison of Implementations

The first experiment consisted of running the implementations of the Prefix-Sums [41, 58] and the parallel Breadth-First Search algorithms discussed in sections 1.7 and 1.8 using the XMT simulator.

Figure 1.5.a presents the results reported for k -ary prefix-sums ($k = 2$) by the XMT cycle-accurate simulator for input sizes in the range 2,000...20,000. The results show the synchronous program outperforming the no-busy-wait program and the execution times increasing linearly with N . This is in agreement with the analytic model results in Figure 1.6. For lack of space, we had to defer the text leading to that figure to section 1.7.

However, there are some differences, as well. For example, the larger gap that the simulator shows between the synchronous program and no-busy-wait execution times. This can be explained by the relatively large overheads of the single-spawn instruction, which will be mitigated in future single-spawn implementations by mechanisms such as sleep-wait.

Figure 1.5.b depicts the number of cycles obtained by running two XMTC implementations of the Breadth First Search algorithm, Single-Spawn and Flattened, on the simulator. The results show the execution times for only one iteration of the BFS algorithm, i.e., building BFS tree level $L(i+1)$ given level $L(i)$. To generate the graphs, we pick a value M and choose the degrees of the vertices uniformly at random from the range $[M/2, 3M/2]$, which gives a total number of edges traversed of $|E(i)| = M * N(i)$ on average. Only the total number of edges is shown in Figure 1.7. Another arbitrary choice was to set $N(i+1) = N(i)$, which gives gatekeeper queuing delay (GQD) of $\frac{|E(i)|}{N(i)} = M$ on average.

The number of vertices in levels $L(i)$ and $L(i+1)$ was set to 500. By varying the

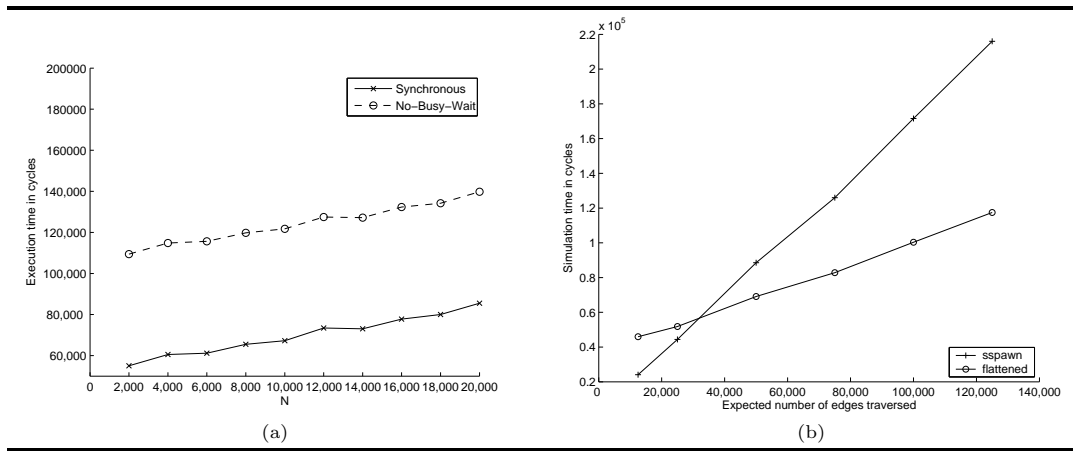


FIGURE 1.5: Cycle counts reported by the XMT cycle-accurate simulator. (a) Synchronous and No-Busy-Wait Prefix-Sums; (b) Flattened and Single-Spawn BFS.

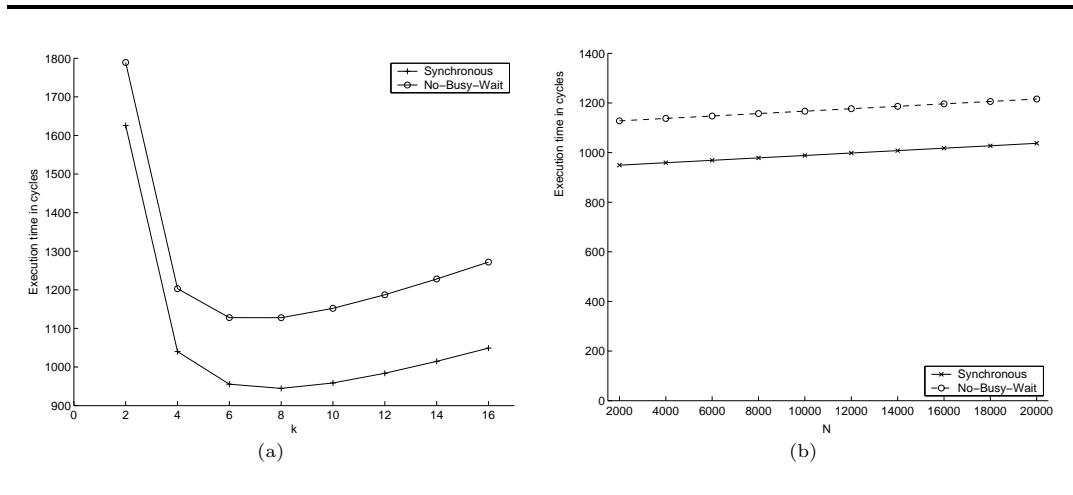


FIGURE 1.6: Estimated run times obtained using the analytic model: (a) Determining the optimum arity of the tree k for the two implementations of the Prefix-Sums algorithm for $N = 1024$. (b) Execution times for the two implementations of the k -ary Prefix-Sums algorithms. The optimum k is chosen for each case.

average degree per vertex M , we generated graphs with the expected number of edges between $L(i)$ and $L(i+1)$ in the range $[12, 500..125,000]$. We observe that the single-spawn implementation outperforms the Flattened BFS for smaller problem sizes, but the smaller work factor makes the latter run faster when the number of edges traversed increases above a certain threshold.

By comparing these experimental results with the outcome of the analysis presented in

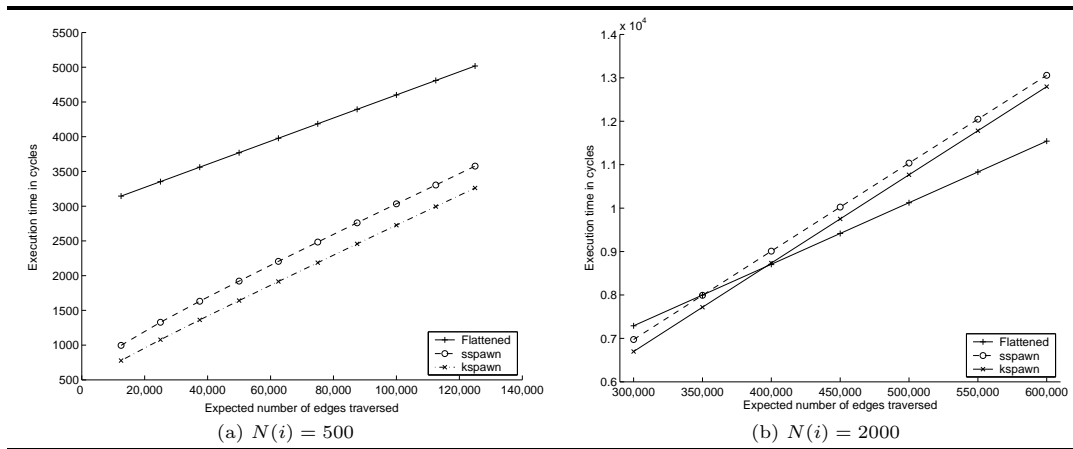


FIGURE 1.7: Analytic execution times for one iteration of BFS when the number of vertices at current level is (a) $N(i) = 500$ and (b) $N(i) = 2000$. The optimal value for k was calculated for each dataset.

Figure 1.7, we observe the same performance ranking between the different implementations providing a second example where the outcome of the analytic performance model is consistent with the cycle-accurate simulations. The text supporting Figure 1.7 is in section 1.9.

1.4.2 Scaling with the Architecture

In this section, we report experiments with the scalability of implementations relative to the number of parallel processing units physically available. Both the XMT simulator and our analytic model are fully parametrized, allowing us to obtain running times for many XMT configurations.

For both analytic and experimental scaling results, we computed the ratio of running times when using one single TCU divided by the time when using 16, 32, 128, 512 or 1024 TCUs. Note that such ratio cancels out constant factors present in the two approaches.

We applied this method to the parallel Sparse Matrix - Dense Vector multiplication (or Matvec) algorithm [24, 51] presented in section 1.12. Results are presented in Table 1.2. The fact that we were able to obtain analytic results that are under 1% different from the simulation cycle-counts for some configurations suggests that the dry analysis can be quite accurate. When the number of TCUs was increased, the larger discrepancy (up to 15.16% for a “full” 1024-TCU XMT configuration) can be explained by the fact that in the analytic model the queuing delay is computed under worst-case assumptions, i.e. proportional to the number of TCUs, while in the simulator the threads are running at different speeds, causing the read requests to spread in time and imply less queuing.

It is worth noting that when scaling down the architecture, the interconnection network becomes simpler, causing the time for a round-trip to first level of cache to decrease; this factor was reflected accordingly in the analytic model.

1.5 Conclusion of Extended Summary

Algorithm	Method	16 TCUs	32 TCUs	128 TCUs	512 TCUs	1024 TCUs
Matvec	Simulation	11.23	22.61	90.32	347.64	646.97
	Analytic	11.42	22.79	84.34	306.15	548.92
	Difference(%)	1.67	0.84	6.63	11.94	15.16

TABLE 1.2 Scalability of algorithms with the number of parallel TCUs

While innovation in education is often not a focus for technical research papers, it is worth noting that the paper provides the missing link for upgrading a standard theoretical PRAM algorithms class to a parallel algorithms and programming class. Specifically, the programming assignments in a one-semester parallel algorithms class taught in 2006 at the University of Maryland included parallel MATVEC, general deterministic (Bitonic) sort, sample sort, breadth first search on graphs and parallel graph connectivity. A first class on serial algorithms coupled with serial programming typically does not require more demanding programming assignments. This is evidence that the PRAM theory coupled with XMT programming could be on par with serial algorithms and programming. An important purpose of the full paper is to provide an initial draft for augmenting a PRAM algorithms textbook with an understanding of how to effectively program an XMT computer system to allow such teaching elsewhere. On the tool-chain side, the following progress has been made: (i) a much more stable compiler has been developed [56]; (ii) from a synthesizable simulator written in the Verilog hardware description language, a cycle-accurate simulator in Java was derived; while more portable and not as slow, such simulators tend to still be rather slow; and (iii) a 75MHz 64-TCU computer based on an FPGA programmable hardware platform has been built using a board comprising 3 Xilinx FPGA chips [61]. This XMT computer is tens of thousands times faster than the simulator. The XMT computer could potentially have a profound psychological effect by asserting that XMT is real and not just a theoretical dream. However, it is also important to realize that for some computer architecture aspects (e.g., relative speed of processors and main memory) a simulator may actually better reflect the anticipated performance of hard-wired (ASIC) hardware that will hopefully be built in the near future.

More generally: (i) The paper suggests that given a proper chip multiprocessor design, parallel programming in the emerging multi-core era does not need to be significantly more difficult than serial programming. If true, a powerful answer to the so-called parallel programming open problem is being provided. This open problem is currently the main stumbling block for the industry in getting the upcoming generation of multi-core architectures to improve single task completion time using easy-to-program application programmer interfaces. Known constraints of this open problem, such as backwards compatibility on serial code, are also addressed by the overall approach. (ii) The paper seeks to shed new light on some of the more robust aspects of the new world of chip-multiprocessing as we have envisioned it, and open new opportunities for teaching, research and practice. This work-in-progress is a significant chapter in an unfolding story.

Part II. Discussion, Examples and Extensions

1.6 Compiler Optimizations

Given a program in the XMT Programming Model, an optimizing compiler can perform various transformations on it to better fit the target XMT Execution Model and reduce execution time. We describe several possible optimizations and demonstrate their effect using the Summation algorithm described above.

1.6.1 Nested Parallel Sections

Quite a few PRAM algorithms can be expressed with greater clarity and conciseness when nested parallelism is allowed [10]. For this reason, it is desired to allow nesting parallel sections with arbitrary numbers of threads in the XMT Programming Model. Unfortunately, hardware implementation of nesting comes at a cost. The performance programmer needs to be aware of the implementation overheads. To explain a key implementation problem we need to review the *hardware mechanism that allocates threads to the p physical TCUs*. Consider an SMPD parallel code section that starts with a `spawn(1,n)` command, and each of the n threads ends with a `join` command without any nested spawns. As noted before, the Master TCU broadcasts the parallel code section to all p TCUs. In addition it broadcasts the number n to all TCUs. TCU i , $1 \leq i \leq p$, will check whether $i > n$, and if not it will execute thread i ; once TCU i hits a `join`, it will execute a special “system” `ps` instruction with an increment of 1 relative to a counter that includes the number of threads started so far; denote the result it gets back by j ; if $j > n$ TCU i is done; if not TCU i executes thread j ; this process is repeated each time a TCU hits a `join` until all TCUs are done, when a transition back into serial mode occurs.

Allowing nesting of `spawn` blocks would require: (i) Upgrading this thread allocation mechanism; the number n representing the total number of threads will be repeatedly updated and broadcast to the TCUs. (ii) Facilitating a way for the parent (spawning) thread to pass initialization data to each child (spawned) thread.

In our prototype XMT Programming Model, we allow nested spawns of a small fixed number of threads through the single-spawn and k -spawn instructions; `sspawn` starts one single additional thread while `kspawn` starts exactly k threads, where k is a small constant (such as 2 or 4). Each of these instructions causes a delay of one RTM before the parent can proceed, and an additional delay of 1-2 RTMs before the child thread can proceed (or actually get started). Note that the threads created with a `sspawn` or `kspawn` command will execute the same code as the original threads, starting with the first instruction in the current parallel section. Suppose that a parent thread wants to create another thread whose virtual thread number (as referenced from the SPMD code) is v . First, the parent uses a system `ps` instruction to a global thread-counter register to create a unique thread ID i for the child. The parent then enters the value v in $A(i)$, where A is a specially designated array in memory. As a result of executing an `sspawn` or `kspawn` command by the parent thread: (i) n will be incremented, and at some point in the future (ii) the thread allocation mechanism will generate virtual thread i . The program for thread i starts with reading v through $A(i)$. It then can be programmed to use v as its “effective” thread ID.

An algorithm that could benefit from nested spawns is the BFS algorithm. Each iteration

of the algorithm takes as input L_{i-1} , the set of vertices whose distance from starting vertex s is $i - 1$ and outputs L_i . As noted in section 1.2.2, a simple way to do this is to spawn one thread for each vertex in L_{i-1} , and have each thread spawn as many threads as the number of its edges.

In the BFS example, the parent thread needs to pass information, such as which edge to traverse, to child threads. To pass data to the child, the parent writes data in memory at locations indexed by the child's ID, using non-blocking writes (namely, the parent sends out a write request, and can proceed immediately to its next instruction without waiting for any confirmation). Since it is possible that the child tries to read this data before it is available, it should be possible to recognize that the data is not yet there and wait until it is committed to memory. One possible solution for that is described in the next paragraph. The `kspawn` instruction uses a prefix-sum instruction with increment k to get k thread IDs and proceeds similarly; the delays on the parent and children threads are similar, though a few additional cycles being required for the parent to initialize the data for all k children.

When starting threads using single-spawn or k -spawn, a synchronization step between the parent and the child is necessary to ensure the proper initialization of the latter. Since we would rather not use a “busy-wait” synchronization technique that could overload the interconnection network and waste power, our envisioned XMT architecture would include a special primitive, called *sleep-waiting*: the memory system holds the read request from the child thread until the data is actually committed by the parent thread, and only then satisfies the request.

When advancing from the programming to the execution model, a compiler can automatically transform a nested spawn of n threads, and n can be any number, into a recursive application of single-spawns (or k -spawns). The recursive application divides much of the task of spawning n thread among the newly spawned threads. When a thread starts a new child, it assigns to it half (or $\frac{1}{k+1}$ for k -spawn) of the $n - 1$ remaining threads that need to be spawned. This process proceeds in a recursive manner.

1.6.2 Clustering

The XMT Programming Model allows spawning an arbitrary number of virtual threads, but the architecture has only a limited number of TCUs to run these threads. In the progression from the Programming Model to the Execution Model, we often need to make a choice between two options: (i) spawn fewer threads each effectively executing several shorter threads, and (ii) run the shorter threads as is. Combining short threads into a longer thread is called clustering and offers several advantages: (a) *reduce RTMs and QDs*: we can pipeline memory accesses that had previously been in separate threads; this can reduce extra costs from serialization of RTMs and QDs that are not on the critical path; (b) *(reduce thread initiation overhead*: spawning fewer threads means reducing thread initiation overheads, i.e. the time required to start a new thread on a recently freed TCU; (c) *reduce memory needed*: each spawned thread (even those that are waiting for a TCU) usually takes up space in the system memory to store initialization and local data.

Note that if the code provides fewer threads than the hardware can support, there are few advantages if any to using fewer longer threads. Also, running fewer, longer threads can adversely affect the automatic load balancing mechanism. Thus, as discussed below, the granularity of the clustering is an issue that needs to be addressed.

In some cases, clustering can be used to group the work of several threads and execute this work using a serial algorithm. For example, in the Summation algorithm the elements of the input array are placed in the leaves of a k -ary tree, and the algorithm climbs the tree computing for each node the sum of its children. However, we can instead start with an

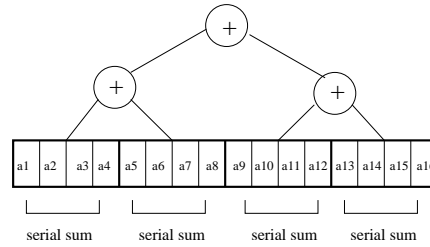


FIGURE 1.8: The sums algorithm with thread clustering.

embarrassingly parallel algorithm in which we spawn p threads that each serially sum $\frac{N}{p}$ elements and then sum the p sums using the parallel summation algorithm. See Figure 1.8.

With such switch to a serial algorithm, clustering is nothing more than a special case of the accelerating cascades technique [16]. For applying accelerating cascades, two algorithms that solve the same problem are used. One of the algorithms is slower than the other, but requires less work. If the slower algorithm progresses in iterations where each iteration reduces the size of the problem considered, the two algorithms can be assembled into a single algorithm for the original problem as follows: 1. start with the slower algorithm and 2. switch to the faster one once the input size is below some threshold. This often leads to faster execution than by each of the algorithms separately.

Finding the optimal crossover point between the slow (e.g., serial) and faster algorithms is needed. Also, accelerating cascades can be generalized to situations where more than two algorithms exist for the problem at hand.

Though our current compiler does not yet do that, a compiler should some day be able to do the clustering automatically. When the number of threads is known statically (i.e., where there are no nested spawns), clustering is simpler. However, even with nested spawns, our limited experience is that methods of clustering tend not to be too difficult to implement. Both cases are described below.

Clustering without Nested Spawns. Suppose we want to spawn N threads, where $N \gg p$. Instead of spawning each as a separate thread, we could trivially spawn only c threads, where c is a function of the number of TCUs, and have each one complete $\frac{N}{c}$ threads in a serial manner. Sometimes an alternative serial algorithm can replace running the N/c threads. Applying this mechanism can create a situation where most TCUs have already finished, but a few long threads are still running. To avoid this, shorter threads can be ran as execution progresses toward completion of the parallel section [47].

Clustering for single-spawn and k -spawn. In the hardware, the number of current virtual threads (either running or waiting) is broadcasted to TCUs as it is updated. Assuming some system threshold, each running thread can determine whether the number of (virtual) threads scheduled to run is within a certain range. When a single-spawn is encountered, if below the threshold, the single-spawn is executed; otherwise, the thread enters a temporary spawning suspension mode and continues execution of the original thread; the thread will complete its own work and can also serially do the work of the threads whose spawning it has suspended. However, the suspension decision can be revoked once the number of threads falls below a threshold. If that occurs, then a new thread is single-spawned. Often, half the remaining work is delegated to the new thread. Clustering with k -spawn is similar. If several threads complete at the same time it will take some time to reach p running threads again, causing a gap in parallel hardware usage. This can be avoided by having a larger threshold, which would keep a set of threads ready to be started as soon

as hardware becomes available. Related implementation considerations have been reported by the MIT Cilk project. As no original insights are claimed here, we a void for brevity a literature review.

1.6.3 Prefetching

Special data prefetch instructions can be used to issue read requests for data values before they are needed; this can prevent long waits due to memory and interconnection network latencies. Prefetched values are stored in read-only buffers at the cluster level. Based on our experiments with different applications, the interconnection network between TCUs and memory is expected to be powerful enough to serve all read requests but perhaps not all prefetch requests. In particular, this suggests avoiding speculative prefetches.

Advanced prefetch capabilities are supported by modern serial compilers and architectures, and the parallel domain is adopting them as well. Prefetching has been demonstrated to improve performance on SMPs [55, 25]. Pai and Adve [48] advocate both grouping read misses and using prefetch. Our approach builds on these results, using thread clustering to group large numbers of read requests, and possibly prefetching them as well. Grouping read requests allows overlapping memory latencies.

1.7 Prefix-Sums

Computing the prefix-sums for n values is a basic routine underlying many parallel algorithms. Given an array $A[0..n-1]$ as input, let $prefix_sum[j] = \sum_{i=0}^{j-1} A[i]$ for j between 1 and n and $prefix_sum[0] = 0$. Two prefix-sums implementation approaches are presented and compared: The first algorithm considered is closely tied to the synchronous (“text-book”) PRAM prefix-sums algorithm while the second one uses a no-busy-wait paradigm [58]. The main purpose of the current section is to demonstrate designs of efficient XMT implementation and the reasoning that such a design may require. It is perhaps a strength of the modeling in the current paper that it provides a common platform for evaluating rather different algorithms. Interestingly enough, our analysis suggests that when it comes to addressing the most time consuming elements in the computation, they are actually quite similar.

Due to [41], the basic routine works in two stages each taking $O(\log n)$ time. The first stage is the Summation algorithm presented previously, namely the computation advances up a balanced tree computing sums. The second stage advances from root to leaves. Each internal node has a value $C(i)$, where $C(i)$ is the prefix-sum of its rightmost descendant leaf. The $C(i)$ value of the root is the sum computed in the first stage, and the $C(i)$ for other nodes is computed recursively. Assuming that the tree is binary, any right child inherits the $C(i)$ value from its parent, and any left child takes $C(i)$ equal to the $C(i)$ of its left uncle plus this child’s value of sum. The values of $C(i)$ for the leaves are the desired prefix-sums. See Figure 1.9.a.

1.7.1 Synchronous Prefix-Sums

The implementation of this algorithm in the XMT Programming model is presented in Table 1.1.c using XMTC pseudocode. Similar to the Summation algorithm, we use a k -ary tree instead of a binary one. The two overlapped k -ary trees are stored using two one-dimensional arrays sum and $prefix_sum$ by using the array representation of a complete tree as discussed in section 1.3.

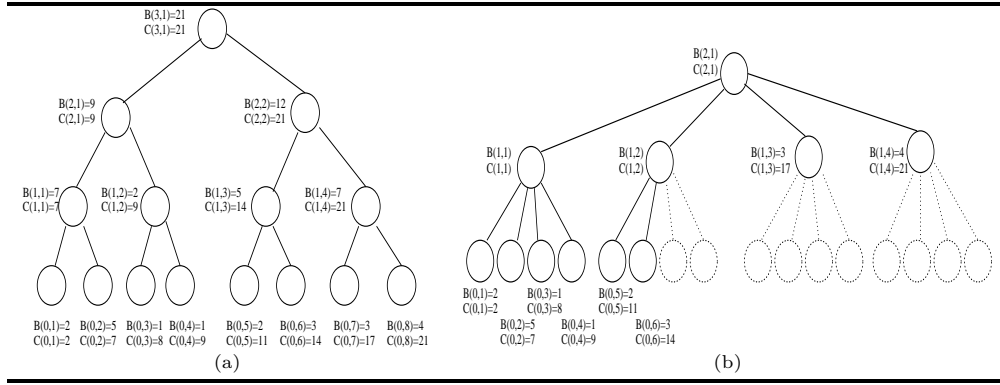


FIGURE 1.9: (a) PRAM prefix-sums algorithm on a binary tree and (b) PRAM prefix-sums algorithm on a k -ary tree ($k=4$).

The XMT algorithm works by first advancing up the tree using a summation algorithm. Then the algorithm advances down the tree to fill in the array *prefix_sum*. The value of *prefix_sum* is defined as follows: (a) for a leaf, *prefix_sum* is the prefix-sum and (b) for an internal node, *prefix_sum* is the prefix-sum for its leftmost descendant leaf (not the rightmost descendant leaf as in the PRAM algorithm - this is a small detail that turns out to make things easier for generalizing from binary to k -ary trees).

Analysis of Synchronous Prefix-Sums

We analyze the algorithm in the XMT Execution Model. The algorithm has 2 round-trips to memory for each level going up the tree. One is to read *sum* from the children of a node, done in one RTM by prefetching all needed values in one round-trip. The other is to join the spawn at the current level. Symmetrically, there are 2 RTMs for each level going down the tree. One to read *prefix_sum* of the parent and *sum* of all the children of a node. Another to join the spawn at the current level. This gives a total of $4 * \log_k N$ RTMs. There is no queuing.

In addition to RTMs, there is a computation cost. The depth is $O(\log_k N)$ due to ascending and descending a logarithmic depth tree. By analyzing our own XMTC implementation we observed this term to be $(7k + 18) \log_k N + 2k + 39$ portion of the depth formula. The *Additional Work* is derived similarly to the summation algorithm. It contains a $\frac{3N}{p}$ term for copying data to the tree's leaves and a $\frac{C * (N - \min(p, N - 1) - 1)}{p} + C * \log_k p$ term to advance up and down the tree. This is derived by using the geometric series to count the number of internal nodes in the tree (because each internal node is touched by one thread and $C = (7k + 4)$ is the work per node) and considering that processing any level of the tree with fewer than p nodes has *Additional Work* = $\frac{(C * p_i)}{p_i} = C$. The overhead to start threads in oversaturated conditions is computed analogously.

For the moment, we do not consider how clustering will be applied. Assuming that a round-trip to memory takes \mathcal{R} cycles, the performance of this implementation is as follows:

$$\begin{aligned}
 \text{Execution Depth} &= (4 \log_k N + 3) \times \mathcal{R} + (7k + 18) \log_k N + 2k + 39 & (1.8) \\
 \text{Additional Work} &= \frac{3N + (7k + 4)(N - \min(p, N - 1) - 1)}{p} +
 \end{aligned}$$

$$+(7k + 4) \log_k p + \left\lceil \frac{(N - \min(p, N - 1) - 1)/(k - 1)}{p} - \log_k \frac{N}{p} \right\rceil \times 2\mathcal{R} \quad (1.9)$$

1.7.2 No-Busy-Wait Prefix-Sums

A less-synchronous XMT algorithm is presented. The Synchronous algorithm presented above processes each level of the tree before moving to the next, but this algorithm has no such restriction. The algorithm is based on the No-Busy-Wait balanced tree paradigm [58]. As before, we use k -ary rather than binary trees.

The input and data structures are the same as previously, with the addition of the *gatekeeper* array, providing a “gatekeeper” variable per tree node. The computation advances up the tree using a No-Busy-Wait summation algorithm. Then it advances down the tree using a No-Busy-Wait algorithm to fill in the prefix-sums.

The pseudocode of the algorithm in the XMT Programming Model is as follows.

```

Spawn(first_leaf , last_leaf )
  Do while alive
    Perform psm on parent's gatekeeper
    If last to arrive at parent
      Move to parent and sum values from children
    Else
      Join
    If at root
      Join
  prefix_sum[0] = 0 //set prefix_sum of root to 0
  Spawn(1,1) //spawn one thread at the root
  Let prefix_sum value of left child = prefix_sum of parent
  Proceed through children left to right where each child is
  assigned prefix_sum value equal to prefix_sum + sum of left
  sibling Use a nested spawn command to start a thread to
  recursively handle each child thread except the leftmost.
  Advance to leftmost child and repeat.

```

Analysis of No-Busy-Wait Prefix-Sums

When climbing the tree, the implementation executes 2 RTMs per level, just as in the previous algorithm. One RTM is to read values of *sum* from the children, and the other is to use an atomic Prefix-sum instruction on the gatekeeper. The LSRTM to descend the tree is also 2 RTMs per level. First, a thread reads the thread ID assigned to it by the parent thread, in one RTM. The second RTM is used to read *prefix_sum* from the parent and *sum* from the children in order to do the necessary calculations. This is an LSRTM of $4 \log_k N$. Also, there are additional $O(1)$ RTMs. Examining our own XMTC implementation, we have computed the constants involved.

Queuing is also a factor. In the current algorithm, up to k threads can perform a prefix-sum-to-memory operation concurrently on the same gatekeeper and create a $k - 1$ queuing delay (since the first access does not count towards queuing delay). The total QD on the critical path is $(k - 1) \log_k N$.

In addition to RTMs and QD, we count computation depth and work. The computation depth is $O(\log_k N)$. Counting the constants our implementation yields $(11 + 8k) * \log_k N +$

$2k + 55$. The $\Sigma \frac{Work}{\min(p, p_i)}$ part of the complexity is derived similarly as in the synchronous algorithm. It contains an $\frac{18N}{p}$ term, which is due to copying data to the tree's leaves and also for some additional work at the leaves. There is a $\frac{C*(N-\min(p, N-1)-1)/(k-1)}{p} + C * \log_k p$ term to traverse the tree both up and down. This value is derived by using the geometric series to count the number of internal nodes in the tree and multiplying by the work per internal node ($C = (11 + 8k)$) as well as considering that processing any level of the tree with fewer than p nodes has $\frac{Work}{\min(p, p_i)} = C$. Without considering clustering, the running time is given by:

$$Execution\ Depth = (4 \log_k N + 6) \times \mathcal{R} + (11 + 9k) * \log_k N + 2k + 55 \quad (1.10)$$

$$Additional\ Work = \frac{6 + 18N + (11 + 8k)(N - \min(p, N - 1) - 1)/(k - 1)}{p} + \\ + (11 + 8k) \log_k p + \\ + \left[\frac{(N - \min(p, N - 1) - 1)/(k - 1)}{p} - \log_k \frac{N}{p} \right] \times 2\mathcal{R} \quad (1.11)$$

1.7.3 Clustering for Prefix-sums

Clustering may be added to the Synchronous k-ary prefix-sums algorithm to produce the following algorithm. The algorithm begins with an embarrassingly parallel section, uses the parallel prefix-sums algorithm to combine results, and ends with another embarrassingly parallel section.

1. Let c be a constant.
 Spawn c threads that run the serial summation algorithm on a contiguous sub-array of N/c values from the input array. The threads write the resulting sum values into a temporary array B .
2. Invoke the parallel prefix-sums algorithm on array B .
3. Spawn c threads. Each thread retrieves a prefix-sum value from B . The thread then executes the serial prefix-sum algorithm on the appropriate sub-array of N/c values from the original array.

The No-Busy-Wait prefix-sums algorithm can be clustered in the same way.

We now present the formulas for execution time using clustering. Let c be the number of threads that are spawned in the embarrassingly parallel portion of the algorithm. Let $SerSum$ be the complexity of the serial summation algorithm, $SerPS$ be the complexity of the serial PS algorithm, and $ParPS$ be the complexity of the parallel PS algorithm (dependent on whether the synchronous or No-Busy-Wait is used). The serial sum and prefix-sum algorithms loop over N elements and from the serial code it is derived that $SerSum(N) = 2N + 1 \times \mathcal{R}$ and $SerPS(N) = 3N + 1 \times \mathcal{R}$. The following formula calculates the cost of performing the serial algorithms on a set of $N - c$ elements divided evenly among p processors and then adds the cost of the parallel step:

$$Execution\ Depth = SerSum\left(\frac{N - c}{p}\right) + SerPS\left(\frac{N - c}{p}\right) + ParPS(c) \quad (1.12)$$

Optimal k and Optimal Parallel-Serial Crossover

The value c , where $p \leq c \leq N$, that minimizes the formula determines the best crossover point for clustering. Let us say $p = 1024$ and $\mathcal{R} = 24$. Similar to the Summation problem,

we have concluded that in the XMT Execution Model for many values $N \geq p$, the best c is 1024. This is the case for both algorithms. A different value of c may be optimal for other applications, for example if the threads do not have equal work.

The optimal k value, where k denotes the arity of the tree, to use for either of the prefix-sums algorithms can be derived from the formulas. As shown in Figure 1.6.a, for $N \geq p$ (where $p = 1024$), the parallel sums algorithm is only run on $c = 1024$ elements and in this case $k = 8$ is optimal for the synchronous algorithm and $k = 7$ is optimal for the No-Busy-Wait algorithm. When $N < p$, clustering does not take effect, and the optimal value of k varies with N , for both algorithms.

1.7.4 Comparing Prefix-sums Algorithms

Using the performance model presented previously with these optimizations allows comparison of the programs in the XMT Execution Model. The execution time for various N was calculated for both prefix-sums algorithms using the formula with clustering. This is plotted in Figure 1.6.b.

The Synchronous algorithm performs better, due to the smaller computation constants. The LSRTM of both algorithms is the same, indicating that using gatekeepers and nesting is equivalent in RTMs to using synchronous methods. The No-Busy-Wait algorithm has slightly longer computation depth and more computation work due to the extra overhead.

We note that in an actual XMT system, an implementation of the Prefix-Sums algorithm would be likely to be included as a library routine.

1.8 Programming Parallel Breadth-First Search Algorithms

As noted earlier, Breadth-First Search (BFS) provides an interesting example for XMT programming. We assume that the graph is provided using the incidence list representation, as pictured in Figure 1.10.a.

Let $L(i)$ be the set of $N(i)$ nodes in level i and $E(i)$ the set of edges adjacent to these nodes. For brevity, we will only illustrate how to implement one iteration. Developing the full program based on this is straightforward.

As described in section 1.2.2, the High-Level Work-Depth presentation of the algorithm starts with all the nodes in parallel, and then using nested parallelism ramps up more parallelism to traverse all their adjacent edges in one step. Depending on the extent that the target programming model supports nested parallelism, the programmer needs to consider different implementations. We discuss these choices in the following paragraphs, laying out assumptions regarding the target XMT model.

We noted before that the Work-Depth model is not a direct match for our proposed programming model. With this in mind, we will not present a full Work-Depth description of the BFS algorithm; as will be shown, the “ideal” implementation will be closer to the High-Level Work-Depth presentation.

1.8.1 Nested Spawn BFS

In a XMT programming model that supports nested parallel sections, the High-level XMT program can be easily derived from the HLWD description:

```

For every vertex v of current layer L(i) spawn a thread
    For every edge e=(v,w) adjacent on v spawn a thread
        Traverse edge e
  
```

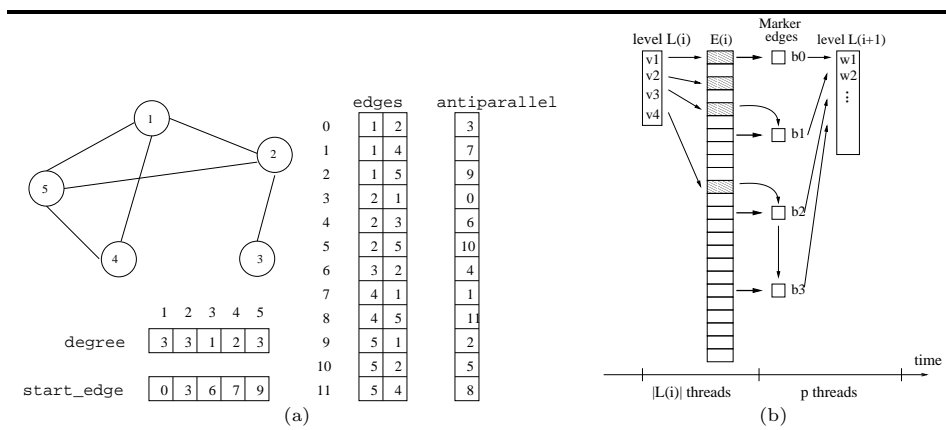


FIGURE 1.10: (a) The incidence list representation for a graph. (b) Execution of Flattened BFS algorithm. First allocate $E[i]$ to hold all edges adjacent to $level[i]$. Next, identify marker edges b_i , which give the first edge per each sub-array. Running one thread per sub-array, all edges are traversed to build $level[i + 1]$.

A more detailed implementation of this algorithm using the XMTC programming language is included in Table 1.1.d. To traverse an edge, threads use an atomic prefix-sum instruction on a special “gatekeeper” memory location associated with the destination node. All gatekeepers are initially set to 0. Receiving a 0 from the prefix-sum instruction means the thread was the first to reach the destination node. The newly discovered neighbors are added to layer $L(i + 1)$ using another prefix-sum operation on the size of $L(i + 1)$. In addition, the edge anti-parallel to the one traversed is marked to avoid needlessly traversing it again (in the opposite direction) in later BFS layers. Note that this matches the PRAM Arbitrary Concurrent Write convention, as mentioned in section 1.2.3.

The Nested Spawn algorithm bears a natural resemblance to the HLWD presentation of the BFS algorithm and in this sense, is the ideal algorithm to program. Allowing this type of implementations to be written and efficiently executed is the desired goal of a XMT framework.

Several other XMT BFS algorithms will be presented to demonstrate how BFS could be programmed depending on the quantitative and qualitative characteristics of a XMT implementation.

1.8.2 Flattened BFS

In this algorithm, the total amount of work to process one layer (i.e. the number of edges adjacent to its vertices) is computed, and it is evenly divided among a pre-determined number of threads p , value which depends on architecture parameters. For this, a Prefix-sums subroutine is used to allocate an array of size $|E(i)|$. The edges will be laid out flat in this array, located contiguously by source vertex. p threads are then spawned, each being assigned one sub-array of $|E(i)|/p$ edges and traversing these edges one by one. An illustration of the steps in this algorithm can be found in Figure 1.10.b.

To identify the edges in each sub-array, it is sufficient to find the first (called *marker*) edge in such an interval; we can then use the natural order of the vertices and edges to find the rest. We start by identifying first (if any) marker edge adjacent to v_j for all vertices

$v_j \in L(i)$ in parallel, then use a variant of the pointer jumping technique [37, 59] to identify the rest of the marker edges (if any) adjacent to v_j using at most $\log_2 p$ steps.

1.8.3 Single-spawn and k -spawn BFS

Although the programming model can allow nested parallelism, the execution model might include limited or no support for nesting. To provide insight into the transformations applied by the compiler, and how to reason about the efficiency of the execution, we present two implementations of the Nested Spawn BFS algorithm that directly map into an Execution model with limited support for nesting.

The **Single-spawn BFS Algorithm** uses `sspawn()` and a binary tree type technique to allow the nested spawning of any number T of threads in $\log_2 T$ steps. The algorithm spawns one thread for each vertex in the current level, and then uses each thread to start $\text{degree}(\text{vertex}) - 1$ additional threads by iteratively using the `sspawn()` instruction to delegate half a thread's work to a new thread. When one edge per thread is reached, the edges are traversed.

The pseudo-code for a single layer is as follows.

```

For every vertex v of current layer L spawn a thread
  While a thread needs to handle  $s > 1$  edges
    Use sspawn() to start another thread and
      delegate  $\text{floor}(s/2)$  edges to it
  Traverse one edge

```

The **k -spawn BFS Algorithm** follows the same principle as Single-spawn BFS, but uses the `kspawn()` instruction to start the threads faster. By using a k -ary rather than binary tree to emulate the nesting, the number of steps to start T threads is reduced to $\log_k T$.

The k -spawn BFS pseudo-code for processing one layer is:

```

For every vertex v of current layer L spawn a thread
  While a thread needs to handle  $s > k$  edges
    Use kspawn() to spawn  $k$  threads and
      delegate to each  $\text{floor}(s/(k+1))$  edges
  Traverse (at most)  $k$  edges

```

1.9 Execution of Breadth-First Search Algorithms

In this section, we examine the execution of the Breadth First Search algorithms presented, and analyze the impact compiler optimizations could have on their performance using the XMT Execution Model as a framework to estimate running times.

Flattened BFS

When each of p threads traverses the edges in its sub-array serially, a simple optimization would prefetch the next edge data from memory and overlap the prefix-sum operations, thus reducing the number of round-trips to memory from $O(\frac{|E(i)|}{p})$ to a small constant. Such an improvement can be quite significant.

The Flattened BFS algorithm uses the prefix sums algorithm as a procedure; we will use the running time computed for this routine in section 1.7. Analyzing our implementation, we observed that identifying the marker edges uses 3 RTMs to initialize one marker edge

per vertex, and then $4 \log_2 p$ RTMs to do $\log_2 p$ rounds of pointer jumping and find the rest of the adjacent marker edges. Finally, p threads cycle through their subarrays of $\frac{|E(i)|}{p}$ edges. By using the optimizations described above, the only roundtrip to memory penalties paid in this step are that of traversing a single edge. A queuing delay occurs at the node gatekeeper level if several threads reach the same node simultaneously. This delay depends on the structure of the graph, and is denoted GQD in the formula below.

In addition to LSRTM and QD, the computation depth also appears in the depth formula. The $10 \log_2 p$ term is the computation depth of the binary tree approach to identifying marker edges. The computation depth of the call to prefix-sums is included.

The dominating term of the Additional Work is $7|E(i)|/p + 28N(i)/p$, which comes from the step at the end of the algorithm in which all the edges are traversed in parallel by p threads, and the new found vertices are added to level $L(i + 1)$. The *Additional Work* portion of the complexity also contains the work for the call to prefix-sums. The performance is:

$$\begin{aligned} \text{Execution Depth} &= (4 \log_k N(i) + 4 \log_2 p + 14) \times \mathcal{R} + 38 + 10 \log_2 p + 7|E(i)|/p + \\ &+ 16N(i)/p + GQD + \text{Computation Depth}(\text{Prefix sums}) \end{aligned} \quad (1.13)$$

$$\begin{aligned} \text{Additional Work} &= \frac{7|E(i)| + 28N(i) + 15p + 17}{p} + \lceil \frac{N(i) - p}{p} \rceil \times \mathcal{R} + \\ &+ \text{Additional Work}(\text{Prefix sums}) \end{aligned} \quad (1.14)$$

As before, $N(i)$ is the number of nodes in layer $L(i)$, $|E(i)|$ is the number of edges adjacent to $L(i)$ and \mathcal{R} is the number of cycles in one RTM.

The second term of relation 1.14 denotes the overhead of starting additional threads in over-saturated conditions. In the flattened algorithm, this can occur only in the initial phase, when the set of edges $E(i)$ is filled in. To reduce this overhead, we can apply clustering to the relevant parallel sections.

Note the following special case: when the number of edges adjacent to one layer is relatively small, there is no need to start p threads to traverse them. We choose a threshold θ , and if $\frac{|E(i)|}{p} < \theta$, then we use $p' = \frac{|E(i)|}{\theta}$ threads. Each will process θ edges. In this case, the running time is found by taking the formulas above and replacing p with $p' = \frac{|E(i)|}{\theta}$.

Single-spawn BFS

In this algorithm, one thread per each vertex v_i is started, and each of these threads then repeatedly uses single-spawn to get $\text{deg}(v_i) - 1$ threads started.

To estimate the running time of this algorithm, we proceed to enumerate the operations that take place on the critical path during the execution:

- Start and initialize the original set of $N(i)$ threads, which in our implementation takes 3 RTMs to read the vertex data.
- Let d_{max} be the largest degree among the nodes in current layer. Use single-spawn and $\log_2 d_{max}$ iterations to start d_{max} threads using a balanced binary tree approach. Starting a new thread at each iteration takes 2 RTMs (as described in section 1.6.1), summing up $2 \log_2 d_{max}$ RTM on the critical path.
- The final step of traversing edges implies using one prefix-sum instruction on the gatekeeper location and another one to add the vertex to the new layer.

The cost of queuing at gatekeepers is represented by GQD . In our implementation, the computation depth was $18 + 7 \log_2 d_{max}$.

Up to $|E(i)|$ threads are started using a binary tree, and when this number exceeds the number of TCUs p , we account for the additional work and the thread starting overhead. We estimate these delays by following the same reasoning as with the k -ary Summation algorithm in section 1.3 using a constant of $C = 19$ cycles work per node as implied by our implementation.

The performance is:

$$\text{Execution Depth} = (7 + 2 \log_2 d_{max})\mathcal{R} + (18 + 7 \log_2 d_{max}) + GQD \quad (1.15)$$

$$\begin{aligned} \text{Additional Work} = & \frac{19(|E(i)| - \min(p, |E(i)| - 1) - 1) + 2}{p} + \\ & + 19 \log_2 |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} \end{aligned} \quad (1.16)$$

To avoid starting too many threads, the clustering technique presented in section 1.6.2 can be applied. This will reduce the additional work component since the cost of allocating new threads to TCUs will no longer be paid for every edge.

k -spawn BFS

The threads are now started using k -ary trees and are therefore shorter. The LSRTM is $2 \log_k d_{max}$. The factor of 2 is due to the 2 RTMs per k -spawn, as per Section 1.6.1.

The computation depth in our XMTC implementation is $(5 + 4k) \log_k d_{max}$. This is an $O(k)$ cost per node, where $\log_k d_{max}$ nodes are on the critical path. The queuing cost at the gatekeepers is denoted by GQD . The *Additional Work* is computed as in Single-spawn BFS with the constant $C = 4k + 3$ denoting the work per node in the k -ary tree used to spawn the $|E(i)|$ threads.

The performance is:

$$\text{Execution Depth} = (7 + 2 \log_k d_{max})\mathcal{R} + (5 + 4k) \log_k d_{max} + 15 + 4k + GQD \quad (1.17)$$

$$\begin{aligned} \text{Additional Work} = & \frac{14|E(i)| + (4k + 3)(|E(i)| - \min(p, |E(i)| - 1) - 1)/(k - 1)}{p} + \\ & + (4k + 3) \log_k |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} \end{aligned} \quad (1.18)$$

Similar to the case of the Single-spawn BFS algorithm, thread clustering can be used in the k -spawn BFS algorithm by checking the number of virtual threads to determine whether the k -spawn instruction should continue to be used or if additional spawning is to be temporarily suspended.

Comparison

We calculated execution time for one iteration (i.e., processing one BFS level) of the BFS algorithms presented here and the results are depicted in Figure 1.7. This was done for two values for the number of vertices $N(i)$ in current level $L(i)$, 500 and 2000. The analysis assumes that all edges with one end in $L(i)$ lead to vertices which have not been visited in a previous iteration; since there is more work to be done for a “fresh” vertex, this constitutes a

worst-case analysis. The same graphs are also used in Section 1.4 for empirically computing speedups over serial code as we do not see how they could significantly favor a parallel program over the serial one. To generate the graphs, we pick a value M and choose the degrees of the vertices uniformly at random from the range $[M/2, 3M/2]$, which gives a total number of edges traversed of $|E(i)| = M * N(i)$ on average. Only the total number of edges is shown in Figure 1.7. We arbitrarily set $N(i + 1) = N(i)$, which gives a queuing delay at the gatekeepers (GQD) of $\frac{|E(i)|}{N(i)} = M$ on average. As stated in section 1.2.5, we use the value $\mathcal{R} = 24$ for the number of cycles needed for one roundtrip to memory.

For small problems, the k -spawn algorithm came ahead and the Single-spawn one was second best. For large problems, the Flattened algorithm performs best, followed by k -spawn and Single-spawn. When the hardware is sub-saturated, the k -spawn and Single-spawn algorithms do best because their depth component is short. These algorithms have an advantage on smaller problem sizes due to their lower constant factors. The k -spawn implementation performs better than Single-spawn due to the reduced height of the “Spawn tree”. The Flattened algorithm has a larger constant factor for the number of RTMS, mostly due to the introduction of a setup phase which builds and partitions the array of edges. For super-saturated situations, the Flattened algorithm does best due to a smaller work component than the other algorithms.

Note that using the formulas ignores possible gaps in parallel hardware usage. In a highly unbalanced graph, some nodes have high degree while others have low degree. As many nodes with small degree finish, it may take time before the use of parallel hardware can be ramped up again. For example, in the Single-spawn and k -spawn algorithms, the virtual threads from the small trees can happen to take up all the physical TCUs and prevent the deep tree from getting started. The small trees may all finish before the deep one starts. This means we are paying the work of doing the small trees plus the depth of the deep tree. A possible workaround would be to label threads according to the amount of work they need to accomplish and giving threads with more work a higher priority (e.g. by scheduling them to start as soon as possible). Note that this issue does not affect the Flattened BFS algorithm, since the edges in a layer are evenly distributed among the running threads.

1.10 Adaptive Bitonic Sorting

Bilardi and Nicolau’s [9] Adaptive Bitonic Sorting algorithm is discussed next. The key component of this sorting algorithm is a fast, work-optimal merging algorithm based on Batcher’s bitonic network and adapted for shared memory parallel machines. An efficient, general purpose Merge-Sort type algorithm is derived using this merging procedure.

The main advantages of the Bitonic Merging and Sorting algorithms are the small constants involved and the fact that they can be implemented on an EREW PRAM. This is an important factor for implementing an algorithm on our proposed XMT model, since it guarantees that no queuing delays will occur. We will only consider the case where the problem size is $N = 2^n$; the general case is treated in [9]. We note that XMT caters well to the heavy reliance on pointer manipulation in the algorithm, which tends to be a weakness for other parallel architectures.

For conciseness, we focus on presenting the Merging algorithm here. The purpose of this algorithm is to take two sorted sequences and merge them into one single sorted sequence. To start, one of the input sequences is reversed and concatenated with the other one. The result is what is defined as a *bitonic sequence*, and it is stored in a *bitonic tree* - a fully balanced binary tree of height $\log_2 N$ whose in-order traversal yields the bitonic sequence, plus an extra node (called spare) to store the N th element. The goal of the bitonic merging

algorithm is to transform this tree into a binary tree whose in-order traversal, followed by the spare node, gives the elements in sorted order.

The key observation of the algorithm is that a single traversal of a bitonic tree from the root to the leaves is sufficient to have all the elements in the left subtree of the root smaller than the ones in the right subtree. At each of the $\log_2 N$ steps of one such traversal, one comparison is performed, and at most two pairs of values and pointers are exchanged. After one such traversal, the algorithm is applied recursively on the left and right children of the root, and after $\log_2 N$ rounds (that can be pipelined, as explained below) the leaves are reached and the tree is a binary search tree.

The full description of the algorithm can be found in [9], and can be summarized by the following recursive function, called with the root and spare nodes of the bitonic tree and the direction of sorting (increasing or decreasing) as arguments:

```

procedure bimerge(root, spare, direction)
1. compare root and spare values to find direction of swapping
2. swap root and spare values if necessary
3. pl = root.left, pr = root.right
4. while pr not nil
5.   compare pl, pr
6.   swap values of pl, pr and two subtree pointers if necessary
7.   advance pl, pr toward leaves
   end
8. in parallel run bimerge(root.left, root) and
   bimerge(root.right, spare)
end

```

In this algorithm, lines 4-7 traverse the tree from current height to the leaves in $O(\log N)$ time. The procedure is called recursively in line 8, starting at the next lowest level of the tree. This leads to an overall time of $O(\log^2 N)$.

We call *stage*(k) the set of tree traversals that start at level k (the root being at level 0). There are 2^k parallel calls in such a stage. Call *phase*(0) of a stage the execution of lines 1-3 in the above algorithm, and *phase*(i), $i = 1.. \log(N) - k - 1$ the iterations of lines 4-7.

To obtain a faster algorithm, we note that the traversals of the tree can be pipelined. In general, we can start stage $k + 1$ as soon as stage k has reached its *phase*(2). On a synchronous PRAM model, all stages advance at the same speed and thus they will never overlap; on a less-asynchronous PRAM implementation, such as PRAM-on-chip, this type of lockstep execution can be achieved by switching from parallel to serial mode after each phase. With this modification, the bitonic merging has a running time of $O(\log N)$.

We have experimented with two implementations of the Bitonic Merging algorithm:

Pipelined This is an implementation of the $O(\log N)$ algorithm that pipelines the stages. We start with an active workset containing only one thread for *stage*(0) and run one phase at a time, joining threads after one phase is executed. Every other iteration, we initialize a new stage by adding a corresponding number of threads to the active workset. At the same time, the threads that have reached the leaves are removed from the workset. When the set of threads is empty, the algorithm terminates.

Non-pipelined The previous algorithm has a lock-step type execution, with one synchronization point after each phase. An implementation with fewer synchronization points is evaluated, where all phases of a stage are ran in one single parallel section with no synchronizations, followed by a `join` command and then the next

stage is started. This matches the $O(\log^2 N)$ algorithm described above.

For the limited input sizes we were able to test at this time on the XMT cycle-accurate simulator, the performance of the pipelined version fell behind the simpler non-pipelined version. This is mainly caused by the overheads required by the implementation of pipelining. Namely, some form of added synchronization, such as using a larger number of spawn blocks, was added.

1.11 Shared Memory Sample Sort

The Sample Sort algorithm [36, 49] is a commonly used randomized sorting algorithm designed for multiprocessor architectures; it follows a “decomposition first” pattern, making it a good match for distributed memory machines. Being a randomized algorithm, its running time depends on the output of a random number generator. Sample Sort has been proved to perform well on very large arrays, with high probability.

The idea behind Sample Sort is to find a set of $p - 1$ elements from the array, called *splitters*, which are then used to partition the n input elements into p buckets $bucket_0 \dots bucket_{p-1}$ such that every element in $bucket_i$ is smaller than each element in $bucket_{i+1}$. The buckets are then sorted independently.

One key step in the standard Sample Sort algorithm is the distribution of the elements to the appropriate bucket. This is typically implemented using “one-to-one” and broadcasting communication primitives usually available on multiprocessor architectures. This procedure can create delays due to queuing at the destination processor [11, 49, 22].

In this section we discuss the Shared Memory Sample Sort algorithm, which is an implementation of Sample Sort for shared memory machines. The solution presented here departs slightly from the Sample Sorting algorithm and consists of an CREW PRAM algorithm that is better suited for an XMT implementation.

Let the input be the unsorted array A and let p the number of hardware Thread Control Units (TCUs) available. An overview of the Shared Memory Sample Sort algorithms is as follows:

Step 1. In parallel, a set S of $s \times p$ random elements from the original array A is collected, where p is the number of TCUs available and s is called the oversampling ratio. Sort the array S , using an algorithm that performs well for the size of S (e.g. adaptive bitonic sorting). Select a set of $p - 1$ evenly spaced elements from it into S' : $S' = \{S[s], S[2s], \dots, S[(p-1) \times s]\}$. These elements are the splitters which will be used to partition the elements of A into p sets $bucket_i$, $0 \leq i < p$ as follows: $bucket_0 = \{A[i] \mid A[i] < S'[0]\}$, $bucket_1 = \{A[i] \mid S'[0] < A[i] < S'[1]\}$, \dots , $bucket_{p-1} = \{A[i] \mid S'[p-1] < A[i]\}$.

Step 2. Consider the the input array A divided into p subarrays, $B_0 = A[0, \dots, N/p - 1]$, $B_1 = A[N/p, \dots, 2N/p - 1]$ etc. The i th TCU iterates through the subarray B_i and for each element executes a binary search on the array of splitters S' , for a total of N/p binary searches per TCU. The following quantities are computed: (i) c_{ij} - the number of elements from B_i that belong in $bucket_j$. The c_{ij} make up the matrix C as in Figure 1.11, (ii) $bucket_idx[k]$ - the bucket in which element $A[k]$ belongs. Each element is tagged with such an index and (iii) $serial[k]$ - the number of elements in B_i that belong in $bucket_{bucket_idx[k]}$ but are located before $A[k]$ in B_i .

For example, if $B_0 = [105, 101, 99, 205, 75, 14]$ and we have $S' = [100, 150, \dots]$ as splitters, we will have $c_{0,0} = 3$, $c_{0,1} = 2$ etc., $bucket[0] = 1$, $bucket[1] = 1$ etc. and $serial[0] = 0$, $serial[1] = 1$, $serial[5] = 2$.

Step 3. Compute the Prefix-Sums $ps[i, j]$ for each **column** of the matrix C . For example, $ps[0, j], ps[1, j], \dots, ps[p-1, j]$ are the prefix-sums of $c[0, j], c[1, j], \dots, c[p-1, j]$. In addition,

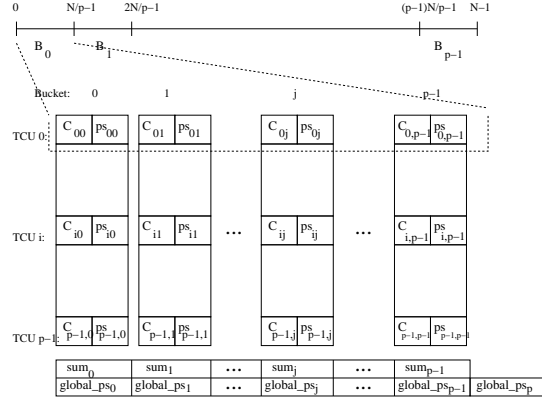


FIGURE 1.11: The helper matrix used in Shared Memory Sample Sort. c_{ij} stores the number of elements from B_i that fall in $bucket_j$. ps_{ij} represent the prefix-sums computed for each column, and $global_ps_{0..p}$ are the prefix-sums and the sum of $sum_{0..p-1}$.

compute the sum of column i , which is stored in sum_i .

Compute the prefix sums of the sum_1, \dots, sum_p into $global_ps[0, \dots, p-1]$ and the total sum of sum_i in $global_ps[p]$.

Step 4. Each TCU i computes: for each element $A[j]$ in segment B_i , $iN/p \leq j < (i+1)N/p-1$:

$$pos[j] = global_ps[bucket[j]] + ps[i, bucket[j]] + serial[j]$$

Copy $Result[pos[j]] = A[j]$.

Step 5. TCU i executes a (serial) sorting algorithm on the elements of $bucket_i$, which are now stored in $Result[global_ps[i], \dots, global_ps[i+1]-1]$.

At the end of Step 5, the elements of A are stored in sorted order in $Result$.

An implementation for the Shared Memory Sample Sort in the XMT programming model can be directly derived from the above description. Our preliminary experimental results show that this sorting algorithm performs well on average; due to the nature of the algorithm, it is only relevant for problem sizes $N \gg p$, and its best performance is for $N \geq p^2$. Current limitations on the cycle-accurate simulator have prevented us from running the sample-sort algorithm on datasets with such a N/p ratio. One possible workaround that we are currently investigating could be to scale down the architecture parameters by reducing the number of TCUs p , and estimate performance by extrapolating the results.

1.12 Sparse Matrix - Dense Vector Multiplication

Sparse matrices, in which a large portion of the elements are zeros, are commonly used in scientific computations. Many software packages used in this domain include specialized functionality to store and perform computation on them. In this section we discuss the so-called *Matvec* problem of multiplying a sparse matrix by a dense vector. Matvec is the kernel of many matrix computations. Parallel implementations of this routine have been used to evaluate the performance of other parallel architectures [24, 51].

To save space, sparse matrices are usually stored in a compact form, for example using a Compressed Sparse Row (CSR) data structure: for a matrix A of size $n \times m$ all the nz

non-zero elements are stored in an array *values*, and two new vectors *rows* and *cols* are used to store the start of each row in *A* and the column index of each non-zero element. An example is shown in Figure 1.12.

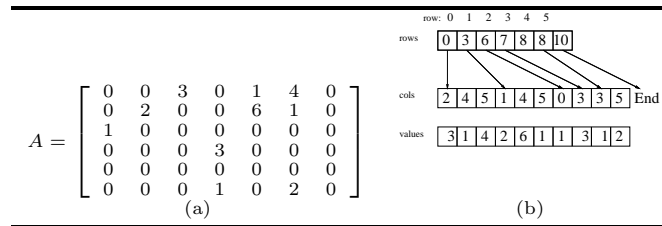


FIGURE 1.12: Compressed Sparse Row representation for a matrix. (a) A sparse matrix. (b) The “packed” CSR representation for matrix *A*.

An “embarrassingly parallel” solution to this problem exists: the rows of the matrix can all be processed in parallel, each being multiplied with the dense vector. If the non-zero elements of the sparse matrix are relatively well distributed among the rows, then the work is well balanced and the algorithm performs well.

One factor that influences the performance of this algorithm on a XMT platform is the fact that the dense vector is read by all the parallel threads. On the XMT architecture, as TCUs request elements of the vector from memory, they are stored in the read-only caches at the cluster level, and subsequent requests from TCUs in the same cluster will not cause round-trips to memory and queuing.

Applying clustering to this algorithm requires a more in-depth analysis, since the threads are no longer symmetrical and can have significantly different lengths. In general, if such information about the lengths of the threads is known before the start of the parallel section, a compiler coupled with a runtime system could use this information to apply thread clustering, as described in section 1.6.2. We note that the total length (number of non-zero elements) and the prefix sums of the number of elements each thread will process is actually provided as part of the input, making this task feasible.

In Spring 2005, an experiment to compare development time between two approaches to parallel programming of Matvec was conducted by software engineering researchers funded by the DARPA (HPCS - High Productivity Computer Systems). One approach was based on MPI and was taught by John Gilbert, a professor at the University of California, Santa Barbara. The second implemented two versions of the above algorithm using XMT in a course taught by Uzi Vishkin at the University of Maryland. Both courses were graduate courses. For the UCSB course this was the fourth programming assignment and for the UMD course it was the second. The main finding [35, 34] was that XMT programming required less than 50% of the time than for MPI programming.

1.13 Speed-ups over Serial execution

We present speed-up results over serial algorithms as observed empirically for some of the applications discussed in this paper. The speed-up coefficients are computed by comparison with a “best serial” algorithm for each application. In our view such comparison is more

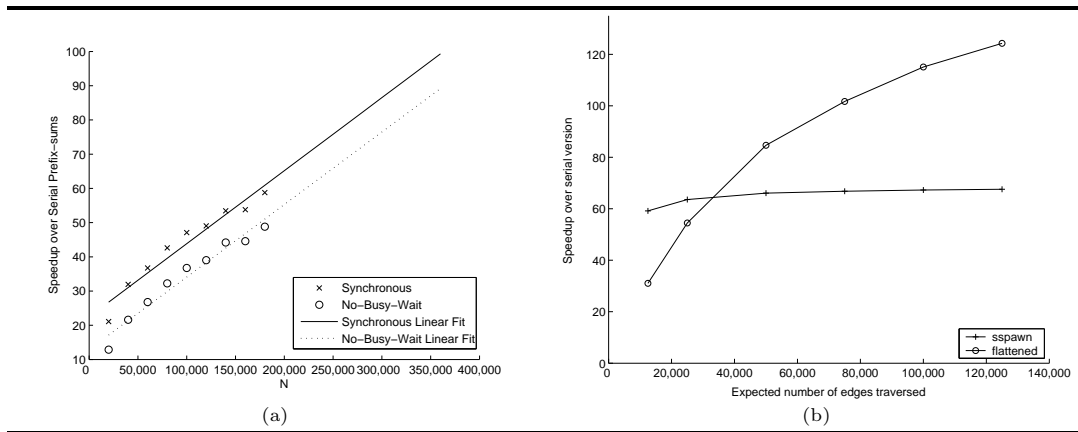


FIGURE 1.13: (a) Speedup over serial prefix-sums. Due to current limitations of our simulator we could only run datasets of size up to 200,000. However, fitting a linear curve to the data indicates that the speedups would exceed 100 for problem sizes above 350,000 and (b) Speedups of BFS algorithms relative to a serial implementation for $N(i) = 500$ vertices in one BFS level.

useful than comparing with the parallel algorithm run on a single processing unit. In order not to give an unfair advantage to the parallel algorithm, we sought to use state-of-the-art compiler optimizations and architectural features when collecting serial running times. To this end, we used the SimpleScalar [14] toolset in the following manner. An “architecture serial speedup coefficient” was computed by running the same serial implementation using two different configurations of the SimpleScalar simulator: (i) one resembling the Master TCU of the XMT platform, and (ii) one using the most advanced architecture simulated by SimpleScalar and aggressive compiler optimizations. We computed coefficients for each application and applied them to calibrate all the speedup results.

To obtain the same speedup results using the analytic model, we would have needed to present a performance model describing a “best serial” architecture and use it to estimate running times for the “best serial” algorithm. We considered this to be outside the scope of this work, and chose to present only experimental speedup results.

Figure 1.13 presents speedup results of parallel Prefix-Sums and Breadth-First Search implementations. The XMT architecture simulated included 1024 parallel TCUs grouped into 64 clusters.

The current paper represents work-in-progress both on the analytic model and on the XMT architecture as a whole. For the future, comparing the outcome of our dry analysis with the experimental results serves a double purpose: (i) It exposes the strengths and limitations of the analytic model, allowing further refinements to more closely match the architecture. (ii) Proposals for new architecture features can be easily evaluated using the analytic model and then confirmed in the simulator. This would be more cost-effective than testing such proposals by committing them to hardware architecture.

1.14 Conclusion

As pointed out earlier, the programming assignments in a one semester parallel algorithms class taught recently at the University of Maryland included parallel MATVEC, general

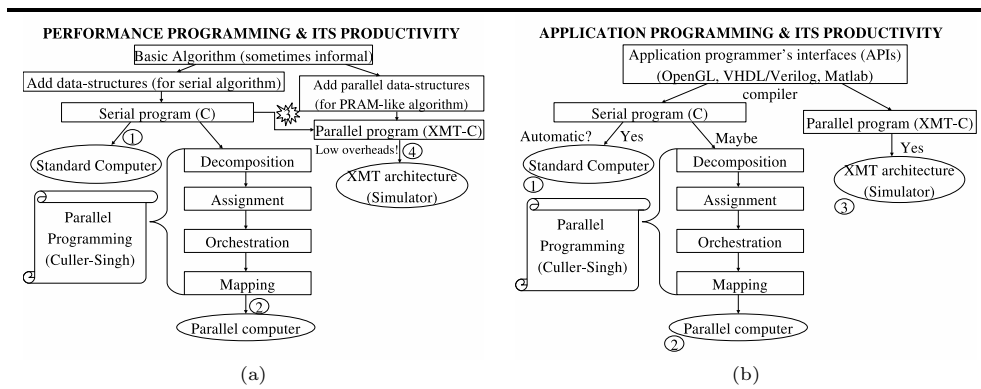


FIGURE 1.14: (a) Productivity of performance programming. Note that the path leading to (4) is much easier than the path leading to (2), and transition (3) is quite problematic. We conjecture that (4) is competitive with (1). (b) Productivity of High-Performance Application Programming. The chain (3) leading to XMT is similar in structure to serial computing (1), unlike the chain to standard parallel computing (2).

deterministic (Bitonic) sort, sample sort, breadth first search on graphs and parallel graph connectivity. A first class on serial algorithms and serial programming typically does not require more demanding programming assignments. This fact provides a powerful demonstration that the PRAM theory coupled with XMT programming are on par with serial algorithms and programming. The purpose of the current paper is to augment a typical textbook understanding of PRAM algorithms with an understanding of how to effectively program a XMT computer system to allow such teaching elsewhere.

It is also interesting to compare the XMT approach with other parallel computing approaches from the point of view of the first course to be taught. Other approaches tend to push the skill of parallel programming ahead of parallel algorithms. In other words, unlike serial computing and the XMT approach, where much of the intellectual effort of programming is taught in algorithms and data structure classes and programming itself is deferred to self-study and homework assignments, the art of fitting a serial algorithm to a parallel programming language such as MPI or OpenMP becomes the main topic. This may explain why parallel programming is currently considered difficult. However, if parallel computing is ever to challenge serial computing as a main stream paradigm, we feel that it should not fall behind serial computing in any aspects and in particular, in the way it is taught to computer science and engineering majors.

Finally, Figure 1.14 gives a bird's eye view on the productivity of both performance programming and application programming (using APIs). By productivity we mean the combination of run time and development time. For performance programming, we contrast the current methodology, where a serial version of an application is first considered and parallelism is then extracted from it using the rather involved methodology outlined for example by Culler and Singh [19], with the XMT approach where the parallel algorithm is the initial target and the way from it to a parallel program is more a matter of skill than an inventive step. For application programming, standard serial execution is automatically derived from APIs. A similar automatic process has already been demonstrated, though much more work remains to be done, for the XMT approach.

Acknowledgments

Contributions and help by the UMD XMT group at and, in particular, N. Ba, A. Balkan, F. Keceli, A. Kupershtok, P. Mazzucco, and X. Wen, as well as the UMD Parallel Algorithms class in 2005 and 2006 are gratefully acknowledged.

References

References

- [1] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. W. H. Freeman & Co., New York, NY, USA, 1994.
- [2] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76, New York, NY, USA, 1987. ACM Press.
- [3] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, 1994.
- [4] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The suif compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, 1995.
- [5] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1988.
- [6] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*, pages 547–556, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] A. Balkan, G. Qu, and U. Vishkin. Mesh-of-trees and alternative interconnection networks for single-chip parallel processing. In *ASAP 2006: 17th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 73–80, Steamboat Springs, Colorado, 2006. Best Paper Award.
- [8] A. O. Balkan and U. Vishkin. Programmer's manual for xmtc language, xmtc compiler and xmt simulator. Technical Report UMIACS-TR 2005-45, University of Maryland Institute for Advanced Computer Studies (UMIACS), February 2006.
- [9] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, 1989.
- [10] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.
- [11] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zaghera. A comparison of sorting algorithms for the connection machine cm-2. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16, New York, NY, USA, 1991. ACM Press.
- [12] R.D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *25th Annual ACM Symposium on the Theory of Computing (STOC '93)*, pages 362–371, 1993.
- [13] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.
- [14] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
- [15] L. Carter, J. Feo, and A. Snively. Performance and programming experience on the tera mta. In *Proceedings SIAM Conference on Parallel Processing*, 1999.
- [16] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro

- and macro techniques for designing parallel algorithms. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219, New York, NY, USA, 1986. ACM Press.
- [17] R. Cole and O. Zajicek. The apram: incorporating asynchrony into the pram model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 169–178, New York, NY, USA, 1989. ACM Press.
 - [18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, first edition, 1990.
 - [19] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
 - [20] S. Dascal and U. Vishkin. Experiments with list ranking for explicit multi-threaded (xmt) instruction parallelism. *J. Exp. Algorithmics*, 5:10, 2000. Special issue for the 3rd Workshop on Algorithms Engineering (WAE'99), London, U.K., July 1999.
 - [21] R. Dementiev, M. Klein, and W. J. Paul. Performance of mp3d on the sb-pram prototype (research note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 132–136, London, UK, 2002. Springer-Verlag.
 - [22] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast parallel sorting under logp: Experience with the cm-5. *IEEE Trans. Parallel Distrib. Syst.*, 7(8):791–805, 1996.
 - [23] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual review of computer science: vol. 3, 1988*, pages 233–283, 1988.
 - [24] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press.
 - [25] M. J. Garzaran, J. L. Briz, P. Ibanez, and V. Vinals. Hardware prefetching in bus-based multiprocessors: pattern characterization and cost-effective hardware. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 345–354, 2001.
 - [26] P. B. Gibbons. A more practical pram model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, New York, NY, USA, 1989. ACM Press.
 - [27] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous pram model. *Theor. Comput. Sci.*, 196(1-2):3–29, 1998.
 - [28] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer: designing a mimd, shared-memory parallel machine (extended abstract). In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, pages 27–42, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.
 - [29] A. Gress and G. Zachmann. Gpu-bisort: Optimal parallel sorting on stream architectures. In *IPDPS '06: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006. To appear.
 - [30] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor. *Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures and Related Research - from System Design to Application Support*, 2006. To appear.
 - [31] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

- [32] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *ALLENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, pages 37–56, London, UK, 1999. Springer-Verlag.
- [33] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [34] L. Hochstein and V. R. Basili. An empirical study to compare two parallel programming models. *18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06)*, July 2006. Position Paper and Brief Announcement.
- [35] L. Hochstein, U. Vishkin, J. Gilbert, and V. Basili. An empirical study to compare the productivity of two parallel programming models. Preprint, 2005.
- [36] J.S. Huang and Y.C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, November 1983.
- [37] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
- [38] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 869–941, 1990.
- [39] J. Keller, C. W. Kessler, and J. L. Traff. *Practical PRAM Programming*. Wiley, New York, NY, USA, 2000.
- [40] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.
- [41] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.
- [42] C.E. Leiserson and H. Prokop. Minicourse on multithreaded programming. <http://supertech.csail.mit.edu/cilk/papers/index.html>, 1998.
- [43] U. Manber. *Introduction to Algorithms - A Creative Approach*. Addison Wesley, 1989.
- [44] K. Moreland and E. Angel. The fft on a gpu. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
- [45] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Evaluating multi-threading in the prototype xmt environment. In *Proceedings 4th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2000.
- [46] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Evaluating the xmt programming model. In *Proceedings of the 6th Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 95–108, 2001.
- [47] D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *invited Special Issue for ACM-SPAA '01: TOCS 36,5*, pages 521–552, New York, NY, USA, 2003. Springer Verlag.
- [48] V. S. Pai and S. V. Adve. Comparing and combining read miss clustering and software prefetching. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 292, Washington, DC, USA, 2001. IEEE Computer Society.
- [49] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.
- [50] J. T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521,

- 1980.
- [51] V. Shah and J. R. Gilbert. Sparse matrices in matlab*p: Design and implementation. In *HiPC*, pages 144–155, 2004.
 - [52] Y. Shiloach and U. Vishkin. An $o(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.
 - [53] S. Skiena. *The Algorithm Design Manual*. Springer, Nov 1997.
 - [54] A. Snaveley, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the tera mta. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.
 - [55] X. Tian, R. Krishnaiyer, H. Saito, M. Girkar, and W. Li. Impact of compiler-based data-prefetching techniques on spec omp application performance. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 53.1, Washington, DC, USA, 2005. IEEE Computer Society.
 - [56] A. Tzannes, R. Barua, G.C. Caragea, and U. Vishkin. Issues in writing a parallel compiler starting from a serial compiler. Draft, 2006.
 - [57] U. Vishkin. From algorithm parallelism to instruction-level parallelism: an encode-decode chain using prefix-sum. In *SPAA '97: Proceedings of the 9th annual ACM symposium on Parallel algorithms and architectures*, pages 260–271, New York, NY, USA, 1997. ACM Press.
 - [58] U. Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 147–155, New York, NY, USA, 2000. ACM Press.
 - [59] U. Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. In use as class notes since 1993. <http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.ps>, February 2002.
 - [60] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 140–151, New York, NY, USA, 1998. ACM Press.
 - [61] X. Wen and U. Vishkin. Pram-on-chip: First commitment to silicon. In preparation, 2007.

APPENDIX: XMTC Code Examples.

K-ary Summation

```

/*
 * void sum(...)
 *
 * The function computes sums by using a k-ary tree.
 * k is defined by the parameter k to the function.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - element 0 of the array is filled with the sum
 *
 */
void sum(int increment[], int result[], int k, int size) {
    register int low, high;
    int height = 20; //note: height should be as large as log_k(size)
    //int layersize[height]; // number of nodes in layer i
    int base[height*size]; // base of leaf is its value after PS
    // base of internal node is the base of its leftmost leaf
    int sum[height*size]; // the value of sum for a node is the sum of the values of
    // increment for all its leaves

    int iteration = 0; // determines the current height in the tree

    int temp;
    int done; //a loop control variable

    int l, //level where leaves start
        sb, //index where leaves would start if size is a power of 2
        d, //size - k^l
        offset, //how much to offset due to size not being power of 2
        sr, //sb + offset
        over, //number of leaves at level l
        under, //number of leaves at level l + 1
        sbp1; //index of one level higher from sb
    int fill; //nodes to fill in with 0 to make all nodes have k children

    int level, startindex, layersize;

    int i;

    /*
     * With non-blocking writes 0 RTM is required to initialize
     * the function parameters: k and size
     * 0 RTM is required to initialize local variables such as height
     */

    //Special case if size == 1
    if(size == 1) { //the check has 0 RTM because size is cached.
        result[0] = 0;
        return;
    }

    /*
     * 18 lines of code above, means computation cost = 18 up to this point.
     */

    //calculate location for leaves in the complete representation
    l = log(size) / log(k);

    sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
    sbp1 = (pow(k,l+1) - 1) / (k - 1);

    d = size - pow(k,l);

```

```

offset = CEIL(((double) d) / (k - 1));
sr = sb + offset;
over = pow(k, l) - offset;
under = size - over;

/*
 * Computation cost = 8
 */

//printf("l = %d, sb = %d, d = %d, offset = %d,
//sr = %d, over = %d\n", l, sb, d, offset, sr, over);

// Copy increment[...] to leaves of sum[...]

low = 0;
high = size - 1;
spawn(low, high) {
    if($ < under) {
        sum[$ + sbp1] = increment[$]; //1 RTM
    }
    else {
        sum[( $ - under) + sb + offset] = increment[$]; //1 RTM
    }
} //1 RTM join

/*
 * LSRTM = 2
 * QD = 0
 * Computation Depth = 5
 * Computation Work = 2N
 */

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
    sum[sbp1 + under + i] = 0;
}

/*
 * Computation Cost = 2k + 1
 */

// Iteration 1: fill in all nodes at level l
low = sb;
high = sb + offset - 1;
if(high >= low) {
    spawn(low, high) {
        int count;

        sum[$] = 0;
        for(count = 0; count < k; count++) {
            sum[$] += sum[k * $ + count + 1];
        }
    }
}

/*
 * We will count the above "Iteration 1" as 1 iteration in
 * the climbing the tree loop below, for simplicity.
 * This gives an upper bound, since the "Iteration 1"
 * section above does slightly less.
 */

// Climb the tree
level = 1;
while(level > 0) {
    level --;
    startindex = (pow(k, level) - 1) / (k - 1);
    layersize = pow(k, level);

    low = startindex;
    high = startindex + layersize - 1;
}

```

```

spawn(low, high) {
    int count;

    /*
     * All the sum[X] elements are read at once
     * for the below loop using prefetching.
     *
     * RTMs = 1
     * (prefetch) Computation depth = k
     */

    sum[$] = 0;
    for(count = 0; count < k; count++) {
        sum[$] += sum[k * $ + count + 1];
    }

    /*
     * Computation Depth = 2k + 1
     */
} // 1 RTM join

/*
 * For the above stage of climbing the tree:
 * LSRTM = 2 * logN
 * Computation Depth = (3k + 9) * logN + 1
 * Computation Work = (3k + 2) * (N - 1) / (k - 1)
 *
 * The (N - 1) / (k - 1) factor of the work is the
 * number of nodes in a k-ary tree of depth logN - 1
 * [there is no work for the leaves at depth logN]
 *
 * Computation Work / min(p, p-i) =
 * ((3k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
 * + (3k + 2) * log_k(p)
 *
 * For each level where number of nodes < p, the denominator is p-i.
 * Otherwise the denominator is p. This gives the above formula.
 */

result[0] = sum[0];

/*
 * For the whole algorithm:
 *
 * LSRTM = 2 * logN + 1
 * QD = 0
 * Computation Depth = (3k + 9) * logN + 2k + 33
 * Computation Work / min(p, p-i) =
 * ((3k + 2)(N - min(p, N-1) - 1) / (k - 1) + 2N) / p
 * + (3k + 2)log_k(p)
 */
}

```

Synchronous K-ary Prefix-Sums

```

/*
 * void kps(...)
 *
 * The function computes prefix sums by using a k-ary tree.
 * k is defined by the parameter k to the function.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * result[] - this array is filled with the prefix sum on the values
 * of the array increment[]
 *
 */

```

```

void kps(int increment[], int result[], int k, int size) {
    register int low, high;
    int height = 20; //note: height should be as large as log_k(size)
    //int layersize[height]; // number of nodes in layer i
    int base[height*size]; // base of leaf is its value after PS
                                // base of internal node is the base of leftmost leaf
    int sum[height*size]; // the value of sum for a node is the sum of the values
                            // of increment for all its leaves

    int iteration = 0; // determines the current height in the tree

    int temp;
    int done; //a loop control variable

    int l, //level where leaves start
        sb, //index where leaves would start if size is a power of 2
        d, //size - k^l
        offset, //how much to offset due to size not being power of 2
        sr, //sb + offset
        over, //number of leaves at level l
        under, //number of leaves at level l + 1
        sbpl; //index of one level higher from sb
    int fill; //nodes to fill in with 0 to make all nodes have k children

    int level, startindex, layersize;

    int i;

    /*
     * With non-blocking writes 0 RTM is required to initialize
     * the function parameters: k and size
     * 0 RTM is required to initialize local variables such as height
     */

    //Special case if size == 1
    if(size == 1) { //the check has 0 RTM because size is cached.
        result[0] = 0;
        return;
    }

    /*
     * 18 lines of code above, means computation cost = 18 up to this point.
     */

    //calculate location for leaves in the complete representation
    l = log(size) / log(k);

    sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
    sbpl = (pow(k,l+1) - 1) / (k - 1);

    d = size - pow(k,l);
    offset = CEIL(((double) d) / (k - 1));
    sr = sb + offset;
    over = pow(k,l) - offset;
    under = size - over;

    /*
     * Computation cost = 8
     */

    //printf("l = %d, sb = %d, d = %d, offset = %d,
    //sr = %d, over = %d\n", l, sb, d, offset, sr, over);

    // Copy increment[...] to leaves of sum[...]

    low = 0;
    high = size - 1;
    spawn(low, high) {
        if($ < under) {
            sum[$ + sbpl] = increment[$]; //1 RTM
        }
        else {
            sum[( $ - under) + sb + offset] = increment[$]; //1 RTM
        }
    }
}

```

```

} //1 RTM join

/*
 * LSRTM = 2
 * QD = 0
 * Computation Depth = 5
 * Computation Work = 2N
 */

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
    sum[sbp1 + under + i] = 0;
}

/*
 * Computation Cost = 2k + 1
 */

// Iteration 1: fill in all nodes at level l
low = sb;
high = sb + offset - 1;
if(high >= low) {
    spawn(low, high) {
        int count;

        sum[$] = 0;
        for(count = 0; count < k; count++) {
            sum[$] += sum[k * $ + count + 1];
        }
    }
}

/*
 * We will count the above "Iteration 1" as 1 iteration in
 * the climbing the tree loop below, for simplicity.
 * This gives an upper bound, since the "Iteration 1"
 * section above does slightly less.
 */

// Climb the tree
level = 1;
while(level > 0) {
    level --;
    startindex = (pow(k, level) - 1) / (k - 1);
    layersize = pow(k, level);

    low = startindex;
    high = startindex + layersize - 1;
    spawn(low, high) {
        int count;

        /*
         * All the sum[X] elements are read at once
         * for the below loop using prefetching.
         *
         * RTMs = 1
         * (prefetch) Computation Depth = k
         */

        sum[$] = 0;
        for(count = 0; count < k; count++) {
            sum[$] += sum[k * $ + count + 1];
        }

        /*
         * Computation Depth of loop = 2k + 1
         */
    }
} // 1 RTM join

/*

```

```

* For the above stage of climbing the tree:
* LSRTM = 2 * logN
* Computation Depth = (3k + 9) * logN + 1
* Computation Work = (3k + 2) * (N - 1) / (k - 1)
*
* The (N - 1) / (k - 1) factor of the work is the
* number of nodes in a k-ary tree of depth logN - 1
* [there is no work for the leaves at depth logN]
*
* Computation Work / min(p, p-i) =
* ((3k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
* + (3k + 2) * log_k(p)
*
* For each level where number of nodes < p, the denominator is p-i.
* Otherwise the denominator is p. This gives the above formula.
*/

base[0] = 0; //set root base = 0

// Descend the tree
startindex = 0;
while(level < 1) {
    layersize = pow(k, level);

    low = startindex;
    high = startindex + layersize - 1;
    spawn(low, high) {
        int count, tempbase;

        tempbase = base[$];

        /*
        * All the sum[X] elements are read at once
        * for the below loop using prefetching.
        *
        * RTMs = 1
        * (prefetch) Computation Depth = k
        */

        for(count = 0; count < k; count++) {
            base[k*$ + count + 1] = tempbase;
            tempbase += sum[k*$ + count + 1];
        }

        /*
        * Computation Depth = 3k;
        */

    } //1 RTM join

    startindex += layersize;
    level++;
}

// Iteration h: fill in all nodes at level l+1
low = sb;
high = sb + offset - 1;
if(high >= low) {
    spawn(low, high) {
        int count, tempbase;

        tempbase = base[$];

        for(count = 0; count < k; count++) {
            base[k*$ + count + 1] = tempbase;
            tempbase += sum[k*$ + count + 1];
        }
    }
}

/*
* For simplicity count "Iteration h" as part of
* the loop to descend the tree. This gives
* an upper bound.
*/

```

```

*
* For the stage of descending the tree:
* LSRTM = 2 * logN
* Computation Depth = (4k + 9) * logN + 2
* Computation Work = (4k + 2) * (N - 1) / (k - 1)
*
* The (N - 1) / (k - 1) factor of the work is the
* number of nodes in a k-ary tree of depth logN - 1
* [there is no work for the nodes at depth logN]
*
* Computation Work / min(p, p-i) =
* ((4k + 2) * (N - min(p, N-1) - 1) / (k - 1)) / p
* + (4k + 2) * log_k p
*
* For each level where number of nodes < p, the denominator is p-i.
* Otherwise the denominator is p. This gives the above formula.
*/

//Copy to result matrix
low = 0;
high = size - 1;
spawn(low, high) {
    result[$] = base[sr + $]; //1 RTM
}

/*
* For above code:
* LSRTM = 1
* Computation Depth = 4
* Computation Work = N
*/

/*
* For the whole algorithm:
*
* LSRTM = 4 * logN + 3
* QD = 0
* Computation Depth = (7k + 18) * logN + 2k + 39
* Computation Work = 3N + (7k + 4) * (N - 1) / (k - 1)
*
* Computation Work / min(p, p-i) =
* (3N + (7k + 4) * (N - min(p, p-i) - 1) / (k - 1)) / p
* + (7k + 4) * log_k p
*/
}

```

No-Busy-Wait K-ary Prefix-Sums

```

/*
* void kps(...)
*
* The function computes prefix sums by using a k-ary tree.
* k is defined by the parameter k to the function.
*
* Input:
* increment[] - an array of increment values
* k - the value of k to use for the k-ary tree
* size - the size of the increment[] array
*
* Output:
* result[] - this array is filled with the prefix sum on
* the values of the array increment[]
*/
void kps(int increment[], int result[], int k, int size) {
    register int low, high;
    int height = 20; //note: height should be as large as log_k(size)
    //int layersize[height]; // number of nodes in layer i
    int base[height*size]; // base of leaf is its value after PS,
    // base of internal node is the base of leftmost leaf
    int sum[height*size]; // the value of sum for a node is the sum
    // of the values of increment for all its leaves
}

```

```

int isLeaf[height * size]; // if a leaf: 1; if not a leaf: 0
int passIndex[height * size]; //array for passing index to child threads

int iteration = 0; // determines the current height in the tree

int temp;
int done; //a loop control variable

int l, //level where leaves start
    sb, //index where leaves would start if size is a power of 2
    d, //size - k^l
    offset, //how much to offset due to size not being power of 2
    sr, //sb + offset
    over, //number of leaves at level l
    under, //number of leaves at level l + 1
    sbpl; //index of one level higher from sb
int fill; //nodes to fill in with 0 to make all nodes have k children

int level, startindex, layersize;

int i;

/*
 * With non-blocking writes 0 RTM is required to initialize
 * the function parameters: k and size
 * 0 RTM is required to initialize local variables such as height
 */

//Special case if size == 1
if(size == 1) { //the check has 0 RTM because size is cached.
    result[0] = 0;
    return;
}

/*
 * 21 lines of code above, means computation cost = 21 up to this point.
 */

//calculate location for leaves in the complete representation
l = log(size) / log(k);

sb = (pow(k,l) - 1) / (k - 1); //this is derived from geometric series
sbpl = (pow(k,l+1) - 1) / (k - 1);

d = size - pow(k,l);
offset = CEIL(((double) d) / (k - 1));
sr = sb + offset;
over = pow(k,l) - offset;
under = size - over;

/*
 * Computation cost = 8
 */

//printf("l = %d, sb = %d, d = %d, offset = %d, sr = %d, over = %d\n", l, sb, d, offset, sr, over)

// Copy increment[...] to leaves of sum[...]

low = 0;
high = size - 1;
spawn(low, high) {
    if($ < under) {
        sum[$ + sbpl] = increment[$]; // 1 RTM
        isLeaf[$ + sbpl] = 1;
    }
    else {
        sum[( $ - under) + sb + offset] = increment[$]; //1 RTM
        isLeaf[( $ - under) + sb + offset] = 1;
    }
} // 1 RTM join

/*
 * For code above:
 */

```

```

* LSRTM = 2
* Computation Depth = 6
* Computation Work = 3N
*/

// Make some 0 leaves at level l+1 so all nodes have exactly
// k children

fill = (k - (under % k)) % k;
for(i = 0; i < fill; i++) {
    sum[sbp1 + under + i] = 0;
}

/*
* Computation Cost = 2k + 1
*/

//Climb tree

low = sr;
high = sr + size + fill - 1;
spawn(low, high) {
    int gate, count, alive;
    int index = $;
    alive = 1;

    while(alive) {
        index = (index - 1) / k;

        gate = 1;
        psm(gate, &gatekeeper[index]); //1 RTM

        if(gate == k - 1) {
            /*
            * Using prefetching, the sum[X] elements
            * in the following loop are read all at once
            * LSRTM = 1
            * (prefetching) Computation Depth = k
            */
            sum[index] = 0;
            for(count = 0; count < k; count++) {
                sum[index] += sum[k*index + count + 1];
            }

            if(index == 0) {
                alive = 0;
            }

            /*
            * Computation Depth = 2k + 3;
            */
        }
        else {
            alive = 0;
        }
    }
} // 1 RTM join

/*
* For code above:
*
* LSRTM = 2 * logN + 1
* QD = k * logN
* Computation Depth = (8 + 2k) * (logN + 1) + 6
* Computation Work = (8 + 2k) * (N - 1) / (k - 1) + 8N
*
* The (N - 1) / (k - 1) factor of the work comes
* from counting the total nodes in a tree with logN - 1
* levels. Each of the leaves at level logN only
* executes the first 8 lines inside the spawn block
* (that is, up to the check of the gatekeeper) before
* most die and only 1 thread per parent continues. This
* gives the 8N term.

```

```

*
* Computation Work / min(p, p-i) =
* ((8 + 2k)*(N - min(p, N-1) - 1)/(k-1) + 8N) / p
* + (8 + 2k) * log_k p
*
* For each level where number of nodes < p, the denominator is p-i.
* Otherwise the denominator is p. This gives the above formula.
*/

base[0] = 0; //set root base = 0

low = 0;
high = 0;
spawn(low, high) {
    int count, tempbase;
    int index = $;
    int newID;

    if($ != 0) {
        index = passIndex[$];
    }

    while(isLeaf[index] == 0) {
        tempbase = base[index];

        /*
        * The k - 1 calls to sspawn can be executed with
        * a single kspawn instruction.
        * The elements sum[X] are read all at once using
        * prefetching.
        *
        * LSRTM = 2
        * (kspawn and prefetching) Computation Depth = k + 1
        */

        for(count = 0; count < k; count++) {
            base[k*index + count + 1] = tempbase;
            tempbase += sum[k*index + count + 1];

            if(count != 0) {
                sspawn(newID) {
                    passIndex[newID] = k*index + count + 1;
                }
            }
        }

        index = k*index + 1;

        /*
        * Computation Depth = 6k + 1
        */
    }
} //1 RTM join

/*
* For code above:
*
* LSRTM = 2 * logN + 1
* Computation Depth = (3 + 6k) * logN + 9
* Computation Work = (3 + 6k) * (N - 1) / (k - 1) + 6N + 6
*
* The (N - 1) / (k - 1) factor of the work comes
* from counting the total nodes in a tree with logN - 1
* levels. Each of the leaves at level logN only
* executes the first 6 lines inside the spawn block
* (up to the check of isLeaf) before dying. This
* gives the 6N term.
*
* Computation Work / min(p, p-i) =
* ((3 + 6k)*(N - min(p, N-1) - 1) / (k-1) + 6N + 6)/p
* + (3 + 6k) * log_k p
*/

//Copy to result matrix

```

```

low = 0;
high = size - 1;
spawn(low, high) {
    result[$] = base[sr + $]; //1 RTM
} //1 RTM join

/*
 * LSRTM = 2
 * Computation Depth = 4
 * Computation Work = N
 */

/*
 * For the whole algorithm:
 *
 * LSRTM = 4 * logN + 6
 * QD = k * logN
 * Computation Depth = (11 + 8k) * logN + 2k + 55
 * Computation Work = (11 + 8k) * (N - 1) / (k - 1) + 18N + 6
 *
 * Computation Work / min(p, p-1) =
 * ((11 + 8k)*(N - min(p, N-1) - 1) / (k-1) + 18N + 6) / p
 * + (11 + 8k)*log_k p
 */
}

```

Serial Summation

```

/*
 * void sum(...)
 * Function computes a sum
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree
 * size - the size of the increment[] array
 *
 * Output:
 * sum
 */
void sum(int increment[], int *sum, int k, int size) {
    int i;
    *sum = 0;

    for(i = 0; i < size; i++) {
        *sum += increment[i];
    }

    /*
     * LSRTM = 1
     * At first, 1 RTM is needed to read increment. However, later reads
     * to increment are accomplished with prefetch.
     *
     * QD = 0
     * Computation = 2N
     */
}

```

Serial Prefix-Sums

```

/*
 * void kps(...)
 *
 * The function computes prefix sums serially.
 *
 * Input:
 * increment[] - an array of increment values
 * k - the value of k to use for the k-ary tree (not used)

```

```

* size - the size of the increment[] array
*
* Output:
* result[] - this array is filled with the prefix sum on the values of the array increment[]
*
*/
void kps(int increment[], int result[], int k, int size) {
    int i;
    int PS = 0;

    for(i = 0; i < size; i++) {
        result[i] = PS;
        PS += increment[i];
    }

    /*
    * LSRTM = 1
    * At first, 1 RTM is needed to read increment. However, later reads
    * to increment are accomplished with prefetch.
    *
    * QD = 0
    * Computation = 3N
    */
}

```

Flattened BFS Algorithm

```

/* Flattened BFS implementation
*/
psBaseReg newLevelGR,notDone; // global register for ps()

int * currentLevelSet, * newLevelSet, *tmpSet; // pointers to vertex sets

main() {

    int currentLevel;
    int currentLevelSize;
    register int low,high;
    int i;
    int nIntervals;

    /* variables for the edgeSet filling algorithm */
    int workPerThread;
    int maxDegree,nMax; // hold info about heaviest node

    /* initialize for first level */
    currentLevel = 0;
    currentLevelSize = 1;
    currentLevelSet = temp1;
    newLevelSet = temp2;

    currentLevelSet[0] = START_NODE;
    level[START_NODE]=0;
    gatekeeper[START_NODE]=1; // mark start node visited

    /* All of the above initializations can be done with non-blocking writes.
    * using 0 RTM
    * 7 lines of code above, cost = 9 up to this point
    */

    // 0 RTM, currentLevelSize in cache
    while (currentLevelSize > 0) { // while we have nodes to explore

        /* clear the markers array so we know which values are uninitialized
        */
        low = 0;
        high = NTCU - 1; // 0 RTM, NTCU in cache
        spawn(low,high) {
            markers[$] = UNINITIALIZED; // 0 RTM, non-blocking write. UNINITIALIZED is a constant
            // the final non-blocking write is overlapped with the RTM of the join
        } // 1 RTM for join

        /* Total for this spawn block + initialization steps before:

```

```

* RTM Time = 1
* Computation time = 1
* Computation work = NTCU, number of TCUs.
*/

/*****
* Step 1:
* Compute prefix sums of the degrees of vertices in current level set
*****/

/*
* We use the k-ary tree Prefix_sums function.
* Changes from "standard" prefix_sums:
* - also computes maximum element. this adds to computation time of
*   upward traversal of k-ary tree
*/

// first get all the degrees in an array
low = 0;
high = currentLevelSize-1;
spawn(low,high) {
  register int LR;
  /* prefetch crtLevelSet[$]
   * this can be overlaped with the ps below,
   * so it takes 0 RTM and 1 computation
   */
  LR = 1;
  ps(LR,GR); // 1 RTM
  degs[GR] = degrees[crtLevelSet[$]];
  // 1 RTM to read degrees[crtLevelSet[$]]. using non-blocking write
  // last write is overlapped with the RTM of the join
} // 1 RTM for join

/* the above spawn block:
* RTM Time = 3
* Computation Time = 3
* Computation Work = 3*Ni
*/

kary_psums_and_max(degs, prefix_sums, k, currentLevelSize, maxDegree);

/*
* this function has:
* RTM Time = 4 log_k (Ni)
* Computation Time = (17 + 9k) log_k (Ni) + 13
* Computation Work = (17 + 9k) Ni + 13
*/
outgoingEdgesSize = prefix_sums[currentLevelSize + 1]; // total sum. 0 RTM (cached)

/* compute work per thread and number of edge intervals
* cost = 3 when problem is large enough, cost = 5 otherwise
* no RTMs, everything is in cache and using non-blocking writes
*/
nIntervals = NTCU; // constant
workPerThread = outgoingEdgesSize / NTCU + 1;
if (workPerThread < THRESHOLD) {
  workPerThread = THRESHOLD;
  nIntervals = (outgoingEdgesSize / workPerThread) + 1;
}
/* Total Step 1:
* RTM Time: 4 log_k Ni + 4
* Computation Time: (17+9k) log_k Ni + 23
* Computation Work: (19+9k) Ni + 21
*/

/*****
* Step 2:
* Apply parallel pointer jumping algorithm to find all marker edges
*****/

nMax = maxDegree / workPerThread; // 0 RTM, all in cache

/* Step 2.1 Pointer jumping - Fill in one entry per vertex */
low = 0;

```

```

// one thread for each node in current layer
high = currentLevelSize - 1;
spawn(low,high) {
  int crtVertex;
  int s,deg;
  int ncrossed;

  /*
   * prefetch currentLevelSet[$], prefix_sums[$]
   * 1 RTM, computation cost = 2
   */

  crtVertex = currentLevelSet[$]; // 0 RTM, value is in cache
  s = prefix_sums[$] / workPerThread + 1; // 0 RTM, values in cache
  // how many (if any) boundaries it crosses.
  ncrossed = (prefix_sums[$] + degrees[crtVertex]) / workPerThread - s;
  // above line has 1 RTM, degrees[] cannot be prefetched above, depends on crtVertex
  if (ncrossed > 0) { // crosses at least one boundary
    markers[s] = s * workPerThread - prefix_sums[$]; // this is the edge index (offset)
    markerNodes[s] = $; // this is the vertex
  }
  // last write is overlapped with the RTM of the join
} // 1 RTM for join

/*
 * Total for the above spawn block
 * RTM Time = 3
 *
 * Computation Time = 9
 * Computation Work <= 9 Ni
 */

/* Step 2.2 Actual pointer jumping */

jump = 1; notDone = 1;
while (notDone) { // is updated in parallel mode, 1 RTM to read it
  notDone = 0; // reset
  low=0; high = NTCU-1;
  spawn(low,high) {
    register int LR;
    // will be broadcasted: jump, workPerThread, UNINITIALIZED constant
    /* Prefetch: markers[$], markers[$-jump]
     * 1 RTM, 2 Computation, 1 QD
     */
    if (markers[$] == UNINITIALIZED) { // 0 RTM, cached
      if (markers[$-jump] != UNINITIALIZED) { // 0 RTM, cached
        // found one initialized marker
        markers[$] = markers[$-jump] + s * workPerThread;
        markerNodes[$] = markerNodes[$-jump];
      }
      else { // marker still not initialized. mark notDone
        LR = 1;
        ps(LR,notDone); // 1 RTM
      }
    }
  }
  // 1 RTM for join
  /* Total for the above spawn block + setup
   * RTM Time = 3
   * Computation time = 6
   * Computation work = 6
   *
   */
  jump = jump * 2; // non-blocking write
}

/* above loop executes at most log NTCU times
 * Total:
 * RTM Time = 4 log NTCU
 * Computation time = 10 log NTCU (includes serial code)
 * Computation work = 6 NTCU
 */

```

```

/* Total step 2:
 * RTM = 4 log NTCU + 3
 * Computation depth = 10 log NTCU + 9
 * Computation work. section 1: 9Ni, section 2=10 NTCU
 */

/*****
 * Step 3.
 * One thread per edge interval.
 * Do work for each edge, add it to new level if new
 *****/

low = 0;
high = nIntervals; // one thread for each interval
newLevelGR = 0; // empty set of nodes
spawn(low, high) {
    int crtEdge, freshNode, antiParEdge;
    int crtNode, i3;
    int gatekLR; // local register for gatekeeper psm
    int newLevelLR; // local register for new level size

    /*
     * Prefetch markerNodes[$], markers[$]
     * 1 RTM, computation cost 2
     */

    crtNodeIdx = markerNodes[$]; // cached, 0 RTM
    crtEdgeOffset = markers[$]; // cached, 0 RTM

    /* prefetch currentLevelSet[crtNodeIdx],
     * vertices[currentLevelSet[crtNodeIdx]],
     * degrees[currentLevelSet[crtNodeIdx]]
     * 2 RTM, cost = 2
     */

    // workPerThread is broadcasted, 0 RTM to read it
    for (i3=0; i3<workPerThread; i3++) {
        crtEdge = vertices[currentLevelSet[crtNodeIdx]] + crtEdgeOffset; // cached, 0 RTM
        // traverse edge and get new vertex
        freshNode = edges[crtEdge][1]; // 1 RTM
        if (freshNode != -1) { // edge could be marked removed
            gatekLR = 1;
            psm(gatekLR, &gatekeeper[freshNode]); // 1 RTM, queuing for the indegree

            if (gatekLR == 0) { // destination vertex unvisited
                newLevelLR = 1;
                // increase size of new level set
                ps(newLevelLR, newLevelGR); // 1 RTM
                // store fresh node in new level. next two lines are 0 RTM, non-blocking writes
                newLevelSet[newLevelLR] = freshNode;
                level[freshNode] = currentLevel + 1;
                // now mark antiparallel edge as deleted
                antiParEdge = antiParallel[crtEdge]; // 0 RTM, prefetched
                edges[antiParEdge][1] = -1; edges[antiParEdge][0] = -1; // 0 RTM, non-blocking writes
            } // end if
        } // end if freshNode

        /* Previous if block costs:
         * 2 RTM, computation 8 for a "fresh" vertex
         * or
         * 1 RTM, computation 2 for a "visited" vertex
         */

        crtEdgeOffset++;
        if (crtEdgeOffset >= degrees[currentLevelSet[crtNodeIdx]]) { // exhausted all the edges?
            // 0 RTM, value is in cache
            crtNodeIdx++;
            crtEdgeOffset = 0;
            /* We have new current node. prefetch its data
             * prefetch currentLevelSet[crtNodeIdx],
             * vertices[currentLevelSet[crtNodeIdx]],
             * degrees[currentLevelSet[crtNodeIdx]]
            */

```

```

        * 2 RTM, cost = 2
        */
    }

    /* This if and instruction before it cost:
    * 2 RTM, 6 computation for each new marker edge in interval
    * or
    * 2 computation for all other edges
    */

    if (crtNodeIdx >= currentLevelSet)
        break;
    // this if is 0 RTM, 1 computation.

} // end for

/* Previous loop is executed  $C = E_i/p$  times.
* We assume  $N_i$  nodes are "fresh", worst case analysis
* Total over all iterations. AA is the number of marker edges in interval.
* WITHOUT PREFETCHING:
* RTM:  $3 * C + 2 AA$ 
* Computation:  $11 * C + 4 AA$ 
*/
// last write is overlapped with the RTM of the join
} // 1 RTM for join

/*
* Total for above spawn block + initialization: ( $C = E_i/p$ ,  $AA = N/p = \#$  marker edges)
* WITHOUT PREFETCHING for multiple edges: RTM Time =  $3 * C + 3 + 2 AA$ 
* WITH PREFETCHING for multiple edges: RTM Time =  $3 + 3 + 2$ 
* Computation Time =  $8 + 7 * C + 16 AA$ 
* Computation Work =  $8p + 7E + 16N$ 
*/

// move to next layer
currentLevel++;
currentLevelSize = newLevelGR; // from the prefix-sums
// "swap" currentLevelSet with newLevelSet
tmpSet = newLevelSet;
newLevelSet = currentLevelSet;
currentLevelSet = tmpSet;

/* all these above steps: 0 RTM, 5 computation */

} // end while
/*
* Total for one BFS level (one iteration of above while loop):
* W/O PRE: RTM Time =  $4 \log_k N_i + 4 |E_i|/p + 11 + \text{LSRTM of PSUMS}$ 
* W PRE : RTM Time =  $4 \log_k N_i + 4 + 11 + \text{LSRTM of PSUMS}$ 
* Computation Time =
* Comp Work =
*/
}

```

Single-Spawn BFS Algorithm

```

/* BFS implementation using single-spawn operation
* for nesting
*/
psBaseReg newLevelGR; // global register for new level set

int * currentLevelSet, * newLevelSet, * tmpSet; // pointers to level sets

main() {

    int currentLevel;
    int currentLevelSize;
    int low, high;
    int i;

    currentLevel = 0;
    currentLevelSize = 1;

```

```

currentLevelSet = temp1;
newLevelSet = temp2;

currentLevelSet[0] = START_NODE; // store the vertex# this thread will handle

/*
 * 0 RTMs, 5 computation
 */

while (currentLevelSize > 0) { // while we have nodes to explore
    newLevelGR = 0;
    low = 0;
    high = currentLevelSize - 1; // one thread for each node in current layer

    spawn(low, high) {
        int gatekLR, newLevelLR, newTID;
        int freshNode, antiParEdge;

        /*
         * All threads need to read their initialization data
         * nForks[$] and currentEdge[$]
         */
        if ($ < currentLevelSize) { // 0 RTM
            /*
             * "Original" threads read it explicitly from the graph
             */
            // only start degree-1 new threads, current thread will handle one edge
            nForks[$] = degrees[currentLevelSet[$]] - 1; // 2 RTM
            // this thread will handle first outgoing edge
            currentEdge[$] = vertices[currentLevelSet[$]]; // 1 RTM
        }
        else {
            /*
             * Single-spawned threads, need to "wait" until init values
             * from the parent are written
             */

            while (locks[$] != 1); // busy wait until it gets the signal
        } // end if

        /* The above if block takes
         * 3 RTM, 3 computation for "original" threads
         * for child threads: 1 RTM for synchronization. 2 computation
         */

        while (nForks[$] > 0) { // 1 computation
            // this is executed for each child thread spawned
            sspawn(newTID) { // 1 RTM
                /*
                 * writing initialization data for child threads.
                 * children will wait till this data is committed
                 */
                nForks[newTID] = (nForks[$]+1)/2 - 1;
                nForks[$] = nForks[$] - nForks[newTID] - 1;
                currentEdge[newTID] = currentEdge[$] + nForks[$]+1;
                locks[newTID] = 1; // GIVE THE GO SIGNAL!

                /*
                 * 0 RTM
                 * 4 computation
                 */
            }
            /* For each child thread:
             * 1 RTM
             * 5 computation
             */
        } // done with forking

        /*
         * Prefetch edges[currentEdge[$]][1], antiParallel[currentEdge[$]]
         * 1 RTM, 2 computation
         */
    }
}

```

```

// let's handle one edge
freshNode = edges[currentEdge[$]][1]; // 0 RTM, value was prefetched
if (freshNode != -1) { // if edge hasn't been deleted

    gatekLR = 1;
    // test gatekeeper
    psm(gatekLR,&gatekeeper[freshNode]); // 1 RTM. GQD queuing

    if (gatekLR == 0) { // destination vertex unvisited!
        newLevelLR = 1;
        // increase size of new level set
        ps(newLevelLR,newLevelGR); // 1 RTM
        // store fresh node in new level
        newLevelSet[newLevelLR] = freshNode;
        level[freshNode] = currentLevel + 1;
        // now mark antiparallel edge as deleted
        antiParEdge = antiParallel[currentEdge[$]]; // 0 RTM, value was prefetched
        edges[antiParEdge][1] = -1;
        edges[antiParEdge][0] = -1;
    } // end if
} // end if

/*
 * Previous if block costs:
 * 2 RTM, 10 computation for "fresh" vertex
 * 0 RTM, 2 computation for visited vertex
 */

/*
 * Final write is blocking, but the RTM overlaps the join.
 */
} // 1 RTM join

/* Computation for a child thread that starts one single child: 19 */

// move to next layer
currentLevel++;
currentLevelSize = newLevelGR; // from the prefix-sums
// "swap" currentLevelSet with newLevelSet
tmpSet = newLevelSet;
newLevelSet = currentLevelSet;
currentLevelSet = tmpSet;

/* the above 5 lines of code: 0 RTM, 5 computation */
} // end while
}

```

k-Spawn BFS Algorithm

The only difference between the single-spawn BFS algorithm and the k-spawn is the while loop that is starting children threads. We're including only that section of the code here, the rest is identical with the code in the BFS Single-Spawn implementation.

```

while (nForks[$] > 0) { // 1 computation
    // this is executed for each child thread spawned
    kspawn(newTID) { // 1 RTM for kSpawn
        // newTID is the lowest of the k TIDs allocated by k-spawn.
        // The other ones are newTID+1, newTID+2,..., newTID+(k-1)
        /*
         * writing initialization data for child threads.
         * children will wait till this data is committed
         */

        slice = nForks[$] / k;
        nForks[$] = nForks[$] - slice; // subtract a slice for parent thread

        for (child=0;child<k;child++) {
            // initialize nForks[newTid + child] and currentEdge[newTid + child]
            nForks[newTID + child] = max(slice,nForks[$]); // for rounding
            currentEdge[newTID] = currentEdge[$] + child * slice;
        }
    }
}

```

```
        nForks[$] = nForks[$] - nForks[newTID + child];
        locks[newTID + child] = 1; // GIVE THE GO SIGNAL!
    }
    /*
    * loop is executed k times.
    * Each iteration:
    * 0 RTM
    * 4 computation
    */
}
/* For each k child threads:
* 1 RTM
* 2+4*k computation
*/
} // done with forking
```