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

Uzi Vishkin George C. Caragea Bryant Lee

April 2006

University of Maryland, College Park, MD 20740

UMIACS-TR 2006-21

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

A bold vision that guided this work is as follows: (i) a parallel algorithms and programming course could become a standard course in every undergraduate computer science program, and (ii) this course could be coupled with a so-called PRAM-On-Chip architecture — a commodity high-end multi-core computer architecture. In fact, the current paper is a tutorial on how to convert PRAM algorithms into efficient PRAM-On-Chip programs. Coupled with a text on PRAM algorithms as well as an available PRAM-On-Chip tool-chain, comprising a compiler and a simulator, the paper provides the missing link for upgrading a standard theoretical PRAM algorithms class to a parallel algorithms and programming class. Having demonstrated that such a course could cover similar programming projects and material to what is covered by a typical first serial algorithms and programming course, the paper suggests that parallel programming in the emerging multi-core era does not need to be 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.

More concretely, a widely used methodology for advancing parallel algorithmic thinking into parallel algorithms is revisited, and is extended into a methodology for advancing parallel algorithms to PRAM-On-Chip programs. A performance cost model for the PRAM-On-Chip 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. Highlighting the importance of LSRTM in determining performance is another contribution of the paper. Finally, some alternatives to PRAM algorithms, which, on one hand, are easier-to-think, but, on the other hand, suppress more architecture details, are also discussed.

1 Introduction

Parallel programming is currently a difficult task. 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. It has been a common

*Partially supported by NSF grant 0325393

sentiment that the development of an easy way for parallel programming would be a major breakthrough; see, e.g., Culler and Singh [CS99].

Indeed, to date the outreach of parallel computing has fallen short of historical expectations. Overall, there is a strong renewed interest in inventing new programming languages that accommodate simple representation of concurrency. However, during the previous decades thousands of papers have been written on this topic. This effort brought about a fierce debate between a considerable number of schools-of-thoughts. One of these approaches, the "PRAM approach", emerged as a clear winner in this "battle of ideas". In fact, we would like to defend an even stronger premise: "Had a parallel architecture that can look to the performance programmer like a PRAM been feasible in the early 1990s, its parallel programming approach would have become common knowledge and the prevailing standard by now". As evidence to support this premise we point out that 3 of the main algorithms textbooks (taught in standard undergraduate computer science courses everywhere by 1990) [Baa88, CLR90, Man89] 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. The only reason that this win did not register in the collective memory as the clear and decisive victory it really is that, at about the same time (early 1990s), it became clear that it will not be possible to build such a machine (i.e., one that can look to the performance programmer as a PRAM) using early 1990s technology.

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. As noted above, the PRAM was developed during the 1980s and 1990s in anticipation of a parallel programmability challenge. It provides the second largest algorithmic knowledge base right next to the standard serial knowledge base.

With the continuing increase of silicon capacity, it becomes possible to build a single-chip parallel processor. Such demonstration has been the purpose of the Explicit Multi-Threading (XMT) project [VDBN98, NNTV03] that seeks to prototype a PRAM-On-Chip vision, as on-chip interconnection networks provide enough bandwidth for connecting processors-to-memories.

Thread-level parallelism (TLP) allows multiple threads of execution to proceed concurrently. There is a long record of compiler efforts for parallelizing serial code. Two representatives include [AALT95, ACK87]. While there have been some success stories, it is now recognized that automatic parallelization by compilers is generally insufficient.

The PRAM-On-Chip platform, to be discussed later in the current paper, is quite broad. The current paper will focus on a thread-level parallelism (TLP) approach for programming it. However, instead of using operating system threads, as in most current systems, threads are defined by the programming language and handled by its implementation. Also, threads are short and the overall objective for multi-threading is reducing single-task completion time.

Several multi-chip multiprocessor architectures targeted implementation of PRAM algorithms, or came close to that: (i) The NYU Ultracomputer project sought to approximate the PRAM [AG94], viewing the PRAM as providing theoretical yardstick for limits of parallelism as opposed to a practical programming model [Sch80]. (ii) The Tera/Cray Multi-threaded Architecture (MTA) advanced Burton Smith's 1978 HEP novel hardware design. Seeking to hide latencies to memory ([SCB+98]) each processor has sufficiently many (128 was a typical number) hardware threads that can context switch quickly. The paper [BCF05] suggests that MTA is close to a PRAM and may allow more efficient implementation of algorithms with irregular memory access such as those from graph theory. Some authors have stated that an MTA with large number of processors looks almost like a PRAM [CFS99]. (iii) The SB-PRAM may be the first multi-chip multiprocessor architecture whose declared objective was to provide emulation of the PRAM [KKT00]. It allows writing computer programs that are similar to the original PRAM algorithms. A 64-processor prototype has been built [DKP02]. (iv) Although a language rather than an architecture, NESL also made a contribution to implementing PRAM algorithms by making the algorithms easier to express using the NESL functional language [Ble96]. NESL programs are compiled and run on standard multi-chip parallel architectures. However, the fact remains that the PRAM theory has generally not reached out beyond the ivory towers of academia. For example, the jury is still out on whether the PRAM can provide an effective abstraction for a proper design of a multi-chip multi-processors. The main difficulty [CS99] appears to be the limits on the bandwidth of such a multi-chip architecture.

More of the case for a lower hanging fruit, PRAM-On-Chip, is presented next. Guided by the fact that the number of transistors on a chip already exceeds one Billion, up from less than 30,000 circa 1980, and keeps growing, the main insight behind PRAM-On-Chip is as follows. The Billion transistor chip era allows for the first time a low-overhead on-chip multi-processor thereby avoiding concerns regarding the higher overhead of multi-chip multiprocessors. It also allows an evolutionary path from serial computing. The drastic recent slow down in clock rate improvement for commodity processors will force vendors to seek single task performance improvements through parallelism. While some have already noted likely growth to 100-core chips by 2015, they are yet to choose programming languages and architectures toward harnessing these enormous hardware resources toward single task completion time. PRAM-On-Chip addresses these issue.

Some key differences between the PRAM-On-Chip 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) PRAM-On-Chip introduced a so-called *Independence of Order Semantics* (IOS), that is 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 2.1). This feature improves performance as it allows processing with whatever data is available at the processing elements and saves power as it reduces synchronization needs. The feature could have been added to multi-chip approaches providing some, but apparently not all the benefits.

Other PRAM-related approaches tended to emphasize competition with (massively parallel) parallel computing approaches and have not paid that much attention to serial code, serial mode in a parallel program, or even parallel execution where the amount of parallelism is low.

The approach could also support standard application programming interfaces (APIs) such as those used for graphics (e.g. OpenGL) or circuit design (e.g. VHDL). Use of high-level APIs can allow automatic extraction of much more parallelism than from code written for performance programming languages such as C. With an effective implementation of such an API for a PRAM-On-Chip (see figure 17.b), an application programmer could take advantage of parallel hardware with few or no changes to an existing API. See [GV06] for a recent example of speedups exceeding a hundred fold over serial computing for gate-level VHDL simulations on PRAM-On-Chip.

The main contribution of this paper is presenting a programming methodology for converting PRAM algorithms to PRAM-on-chip programs. An overview of some alternatives to PRAM algorithms, which are easier-to-think, but, on the other hand, suppress more architecture details, are also discussed. Performance models used in developing a PRAM-On-Chip algorithm are described in section 2. An example of using the models is given in section 3. Section 4 explains compiler optimizations that could affect the actual execution of programs. Section 5 gives another example for applying the models to the prefix sums problem. Section 6 presents Breadth-First Search (BFS) in the PRAM-On-Chip Programming Model. Section 7 explains the application of compiler optimizations to BFS and compares performance of several BFS implementations. Section 8 discusses the Adaptive Bitonic Sorting algorithm and its implementation while section 9 introduces a variant of Sample Sorting that runs on a PRAM-On-Chip. Section 10 discusses matrix-vector multiplication. Some empirical validation of the models is presented in section 11. We conclude in section 12.

2 Model descriptions

Given a problem, a "recipe" for developing an efficient PRAM-on-chip program from concept to implementation is proposed. In particular, the stages through which such development needs to pass are presented.

Figure 1 depicts the proposed methodology. For context, the figure also depicts the widely used Work-

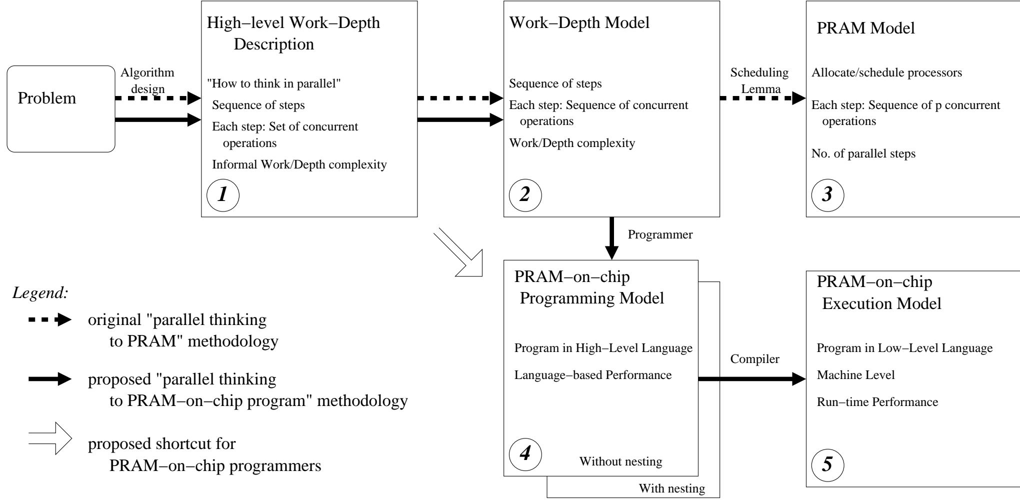


Figure 1: Proposed Methodology for Developing PRAM-On-Chip Programs in view of the Work-Depth Paradigm for Developing PRAM algorithms.

Depth methodology for advancing from concept to a PRAM algorithm; namely, the sequence of models $1 \rightarrow 2 \rightarrow 3$ in the figure illustrates progression from a high-level description to a PRAM algorithm. For developing a PRAM-on-chip implementation, we propose following the sequence of models $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$: given a specific problem, an algorithm design stage will produce a High-Level description of the parallel algorithm; this informal description is fleshed out as a sequence of steps each comprising a set of concurrent operations. In a first draft, the set of concurrent operations can be implicitly defined. See the BFS example in Section 2.2.1. This first draft is refined to a sequence of steps each comprising now a sequence of concurrent operations. Such formal Work-Depth description fully spells out how to advance in a given step, whose sequence of concurrent operations include j operations indexed by integers from 1 to j, from each index i where $1 \leq i \leq j$, to an operation. The programming effort amounts to translating this description into a single-program multiple-data (SPMD) program using a high-level PRAM-on-chip programming language. From this SPMD program, a compiler will transform and reorganize the code to achieve the best performance in the target PRAM-on-chip execution model. As a PRAM-on-chip 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 PRAM-on-chip 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 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.

2.1 PRAM Model

PRAM (for Parallel Random Access Machine, or Model) augments the standard serial model of computation, known as RAM [AU94], with parallelism. A PRAM consists of p synchronous processors and a global shared memory accessible in unit time from each of the processors. The only mean of inter-processor communication is through the shared memory. Different conventions exist regarding concurrent access to the memory, including: (i) exclusive-read exclusive-write (EREW) under which simultaneous access to the same memory location for read or write purposes are forbidden, (ii) concurrent-read exclusive-write (CREW), which allows concurrent reads but not writes, and (iii) concurrent-read concurrent-write (CRCW) where both are permitted, and a convention regarding how concurrent writes are resolved is specified. One of these conventions, Arbitrary CRCW, stipulates that concurrent writes into a common memory location result in an arbitrary processor, among those attempting to write, succeeding, but it is not known in advance which

processor.

The are quite a few sources for PRAM algorithms including [JáJ92, KR90, EG88, Vis02]. 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. Producing such a description imposes a significant burden on the algorithm designer. Luckily this burden can be somewhat mitigated using the Work-Depth methodology.

2.2 The Work-Depth Methodology

Introduced in [SV82], the Work-Depth methodology for designing PRAM algorithms has proved to be quite useful as a framework for describing parallel algorithms and reasoning about their performance. For example, it was used as the description framework in [JáJ92]. The methodology is guided by seeking 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 suggests starting by producing an informal description of the algorithm in a high-level work-depth model (HLWD), and then advancing this description into a fuller presentation in a model of computation called Work-Depth. We proceed to describe these two models next.

2.2.1 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: Given is an undirected graph $G(V, E)$, where the length of every edge in E is 1, and a source node $s \in V$; the *breadth-first search (BFS)* algorithm finds the lengths of the shortest paths from s to every node in V. An informal work-depth description of the parallel BFS algorithm can look as follows. Suppose that V, the set of vertices of the graph G, is partitioned into layers, where layer L_i includes all vertices of V whose shortest path from s includes exactly 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} has already been 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 by 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 try to find an efficient PRAM implementation for an iteration. Namely, given a p-processor PRAM how to allocate processors to tasks to finish all operations of an iterations as quickly as possible? As noted earlier, a more detailed description in the Work-Depth model would address these issues.

2.2.2 Work-Depth Model

In the Work-Depth model the description is to be cast in terms of successive time steps, where the concurrent operations in a time step form a sequence; each element in the sequence is indexed by a different index between 1 and 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 [Bre74] 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. The challenge would be to find a way for listing in a single sequence all the edges that have as an endpoint a vertex of layer L_i. In other words, the Work-Depth model does not allow us to leave nesting of parallelism unresolved. On the other hand PRAM-On-Chip programming should allow nesting since this mechanism provides an easy way for parallel programming. It is also important to note that the PRAM-on-chip architectures includes some limited support for nesting of parallelism. 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") be fully aware of the 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 6, 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 till the compiler matures. It should still need to be taught (even after the ideal compiler solution is in place) in order to explain the cost of nesting to programmers.

The reason for bringing this issue up this early in the discussion is that it actually suggests that our methodology does not necessarily need to make a "complete stop" at the Work-Depth model, but can perhaps detour it and proceed directly to the PRAM-like programming methodology.

2.3 PRAM-on-chip Programming Model

The PRAM-on-chip programming model is a framework for a high-level programming language. It can be used to implement an algorithm described in the Work-Depth presentation model, but as noted before it also offers shortcuts from higher-level descriptions. The overall objective of the programming model is to mitigate two goals: (i) *Programmability:* given an algorithm in HLWD or Work-Depth model, the programmer's effort should be minimized; and (ii) *Implementability:* effective compiler translation into the PRAM-on-chip execution model should be feasible.

A fine-grained, SPMD type model, in which execution frequently alternates between serial and parallel execution mode, is presented. As illustrated in Figure 2, a Spawn command prompts a switch from serial mode to parallel mode. 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: the programmer can use commands (e.g., "prefix-sum") that permit threads to proceed even if they try to write into the same memory location. This was inspired by the PRAM arbitrary concurrent-write convention noted earlier.

The following are some of the primitives in the PRAM-on-chip programming model:

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

Thread-id. A special variable name used inside a parallel section, which evaluates to the thread ID. This allows SPMD style programming.

Prefix-sum Instruction. The prefix-sum instruction defines an atomic operation. Operating on two variables, a base variable B and an increment variable R, the result of a prefix-sum is that B gets the value $B + R$, while R gets the original value of B. Some interesting uses of the prefix-sum instruction are when several concurrent threads use it with respect to the same base. It provides a tool for implementing IOS as well as for inter-thread coordination. While, the basic definition of prefix-sum follows the fetch-and-add of the NYU-Ultracomputer [GGK$^+$82], PRAM-on-Chip uses a fast parallel hardware implementation (`ps()`) if R is from a small range (e.g., one bit) and B can fit one of a small number of global registers; otherwise, prefix-sums are done using a prefix-sum-to-memory (`psm()`) instruction and are resolved by queuing to memory.

Nested parallelism. A parallel thread can be programmed to initiate more threads. However, as noted in Section 2.2.2 this comes with some (tentative) restrictions and cost caveats, due to compiler and

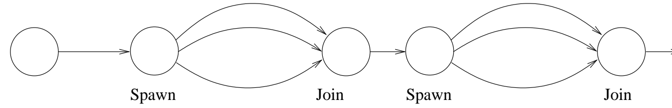


Figure 2: Switching between serial and parallel execution modes in the PRAM-on-chip programming model. Each parallel thread executes at its own speed, without ever needing to synchronize with another thread

hardware support issues. As illustrated with the Breadth-First search example, nesting of parallelism could improve the programmer's ability to describe an algorithms in a clear and concise way. Nesting is discussed in several places in the current paper, including section 4.1.

Note that Figure 1 depicts two alternative PRAM-On-Chip 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.

Since our current embodiment of PRAM-On-Chip is called XMT, for **eXplicit Multi-Threading**, we call the illustration of this programming model XMTC. XMTC is a superset of the language C, obtained from it by adding structures for the above primitives.

Examples of XMTC code Several examples of actual implementations of PRAM algorithms using XMTC are presented in figure 3. While each of these programs is discussed in greater detail in the following sections, the purpose of the table was to convey to readers familiar with other parallel programming frameworks the relative conciseness of these programs. Some language constructs, such as variable and function declarations, have been left out in this table, but they need to be included in a valid XMTC program.

Next, the language features of XMTC are demonstrated using the *array compaction problem*, presented in figure 3.a: given an array of integers $T[0..n-1]$, copy all its non-zero elements into another array S; any order will do. The special variable $ denotes the thread-id. The 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)` instruction 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 global variable which will hold the number of non-zero elements copied at the end of the spawn block. Variables declared inside a `spawn()` block are local for each thread, and are usually much faster to access than the shared memory. [1]

To evaluate performance in this model, a *Language-Based Performance Model* is used: performance costs are assigned to each primitive instruction in the language and rules are specified for combining them into expressions. Such performance modeling was used by Aho and Ullman [AU94] and was generalized for parallelism by Blelloch [Ble96]. The paper [DV00] used language-based modeling for studying parallel list ranking relative to an earlier performance model for XMT.

2.4 PRAM-on-chip Execution Model

The execution model depends heavily on particulars of the PRAM-on-chip implementation. For illustration purposes, we will use the XMT PRAM-on-chip platform (see [NNTV03]).

A bird eye's view of XMT is presented in Figure 4. A number of (say 1024) Thread Control Units (TCUs) are grouped into (say 64) clusters. Clusters are connected to the memory subsystem by a high-throughput, low-latency interconnection network; 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 included at cluster level; this is used to store values read from memory by a TCU and also holds the values read by prefetch instructions. The memory system consists of memory modules each having several levels of cache

[1]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 This type of optimizations can also be performed by an optimizing compiler.

(a) Array compaction

```
length = 0;
spawn(0,n-1) { // start one thread per array element
     int increment = 1;
     if(T[$] != 0) {
       // execute prefix-sum to allocate one entry in array S
          ps(increment, length);
          S[increment] = T[$];
     }
}
```

(b) k-ary Tree Summation

```
/* Input: N numbers in sum[0..N-1]                                              *
 * Output: The sum of the numbers in sum[0]                                     *
 * The sum array is a 1D complete tree representation (See Summation section)  */
level = 0;
while(level < log_k(N) ) { // process levels of tree from leaves to root
        level++;
        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;
        }
}
```

(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--;
}
```

(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]; }
        }
    }
    i++;
}
```

(e) Sparse Matrix - Dense Vector Multiplication

```
/* Input: Vector b[n], sparse matrix A[m][n] given in Compact Sparse Row form, *
 *  as in figure 12                                                            *
 * 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
}
```

Figure 3: Implementation of some PRAM algorithms in the XMT PRAM-on-chip framework to demonstrate compactness.

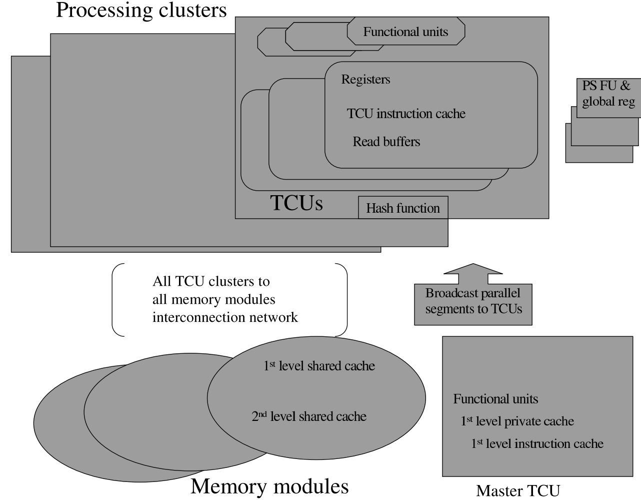


Figure 4: An overview of the XMT PRAM-on-chip Architecture.

memories. In general each logical memory address can reside in only one memory module, alleviating cache coherence problems. This explains why only read-only caches are used at the clusters. The Master TCU runs serial code, or the serial mode for XMT. 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 is based on a thread-id assigned to it. A separate distributed hardware system, reported in [NNTV03] but not shown in figure 4, 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.

A program in the high-level PRAM-on-chip Programming model needs to be translated by an optimizing compiler in order to take advantage of features of the architecture. A program in the Execution model could include prefetch instructions, as well as broadcast instructions, where some values needed by all, or nearly all TCUs, are broadcasted to all. More advanced optimizations such as combining shorter virtual threads into a longer thread (a mechanism called "thread clustering"), are also considered at this optimization stage. 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 4.

To evaluate the performance of a program in this model, we use an extension of the notions of *work* and *depth* to include measurements appropriate for an execution model, and then proceed to give a formula for estimating execution time based on them.

The depth of an application in the PRAM-on-chip 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. A read request 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.

The prefix-sum `ps()` primitive is supported by a special hardware unit that combines `ps()` calls from multiple threads into a single multi-operand prefix-sum operation. In one thread, a `ps()` instruction causes one RTM and 0 queuing delay.

In addition, a prefix-sum to memory (`psm()`) instruction is supported. Its syntax is similar to the `ps()`

instruction except the base variable is a memory location instead of a global register. This instruction is executed by queued updates to the memory location rather than by special hardware, due to the difficulty in creating multi-operand hardware that would operate on arbitrary memory locations. The `psm()` command costs 1 RTM and additionally has a queuing delay equal to the number of threads calling `psm()` on the same location.

We can now define the PRAM-on-chip "execution depth" and "execution time". PRAM-On-Chip 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 PRAM computation depth, LSRTM, and QD on the critical path. Assuming that a round-trip to memory takes $\mathcal{R}$ cycles:

$$Execution\ Depth = Computation\ Depth + LSRTM \times \mathcal{R} + QD \tag{1}$$

Sometimes more *Work* (the total number of instructions executed) can 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 denoted by relation 2. The total time spent executing instructions outside the critical path over all parallel sections is given in relation 3.

$$ThreadStartOverhead_i = \left\lceil \frac{p_i - p}{p} \right\rceil \times \mathcal{R} \tag{2}$$

$$Additional\ Work = \sum_{spawn\ block\ i} \left(\frac{Work_i}{min(p, p_i)} + ThreadStartOverhead_i \right) \tag{3}$$

Adding up, the execution time of the entire program is:

$$Execution\ Time = Execution\ Depth + Additional\ Work \tag{4}$$

2.5 Clarifications of the modeling

Our model of performance attempts to distill the major factors affecting runtime specifically for the PRAM-On-Chip platform. The performance modeling for PRAM-On-Chip has the advantage of being close to the Work-Depth algorithmic framework, with additional accounting for memory costs using the LSRTM and QD.

First, we would like to present a somewhat subtle point: Following the path from the HLWD model to the PRAM-ON-Chip models in Figure 1 may be important not only for the purpose of developing a PRAM-On-Chip program, but also for optimizing performance. Note that bandwidth is not accounted for in the PRAM-On-Chip performance modeling, since a PRAM-On-Chip architecture should be able to provide sufficient bandwidth for an efficient algorithm in the Work-Depth model. In other words, 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.

Let us examine what could happen if PRAM-On-Chip performance modeling is not 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. In other words, 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 PRAM-On-Chip hardware has a limited number of memory modules, and if multiple requests attempt to access the same module, queuing will occur. Although the model accounts for queuing to the same memory location, it does not account for queuing that may occur for accesses to different locations (in the same module). However, 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.

To some limited extent, the effect of these approximations on running times can be observed from the experimental results in section 11, where a comparison with simulations is presented.

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 PRAM-On-Chip 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 note that the number of clocks to access main memory should be about the same as for serial computing and also that both for serial computing and for PRAM-On-Chip large caches can be built. However, this modeling is inappropriate if PRAM-On-Chip 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 it will not be appropriate not to include this figure for cache misses on PRAM-On-Chip, as well.

Note that 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. We could further refine our analysis and propose a more accurate model, but with much more involved modeling. For the sake of clarity, we made the choice to stop at the level of detail that allows for a concise presentation while providing relevant results.

Other researchers that worked on performance modeling of parallel algorithms have typically focused on different factors than those we have identified here. The reason is they dealt with other platforms. Helman and JáJá [HJ99] 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. However, these quantities are less important in a PRAM-like environment. Bader, Cong, and Feo [BCF05] 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, researchers have developed a calculator for performance that includes the parameters of count of trips to memory, number of instructions, and number of accesses to local memory [FHKK05]. Our measures are still different because the RTMs that we count are round trips to the shared cache, and we also count queuing at the shared cache. In addition, 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 PRAM-On-Chip.

3 An Example for Using the Methodology: Summation

Consider the problem of computing the sum of n numbers. Given as input an array A of size n the output provides the sum of its values. Developing a parallel program for this simple problem is presented next as an example for the methodology of the previous section. Progressing through the models is presented. A High-Level Work-Depth description of the algorithm is presented in figure 5.a. A non-recursive Work-Depth presentation of this algorithm can be derived from it, as presented in figure 5.b.

In the WD algorithm, we use an unidimensional array to store all the elements of the tree, as shown in figure 6. For the more general case of a complete k-ary tree, we store the root at element 0, followed by the k elements of the first level, listed from left to right, then the k^2 elements of second level etc. The array is densely packed, with no gaps, thus (a) the children of node i are at indices $k*i+1, k*i+2, \ldots, k*i+k$ and (b) the parent of node i is at index $\lfloor \frac{i-1}{k} \rfloor$. Note that this simple relationship between a node and its children is helpful for improving performance.

We now proceed to express this algorithm in the PRAM-On-Chip Programming Model. Note that the WD algorithm uses a balanced-binary tree approach, by repeatedly adding in parallel pairs of values. Alternatively, k values can be summed serially; this constitutes a k-ary tree approach. The k-ary tree is

(a)	(b)

`SUM(A,n)`
` If n = 1 then sum = A[1]; exit`
` For 1 <= i <= n / 2 pardo`
`          B[i] = A[2i - 1] + A[2i]`
` Call SUM(B,n/2)`
	`For 1 <= i <= n pardo     // B is a 1D array`
	`         B[n-1 + i] = A[i]// model of a tree`
	`For h = logn to 1 do`
	`         For 2^(h-1) <= i < 2^h pardo`
	`                  B[i] = B[2i - 1] + B[2i]`
	`sum = B[1]`

Figure 5: The Summation Algorithm. (a) A High-Level Work Depth presentation. Pairs of values of A are summed up and stored into array B, followed by a recursive call on array B. (b) A Work-Depth description

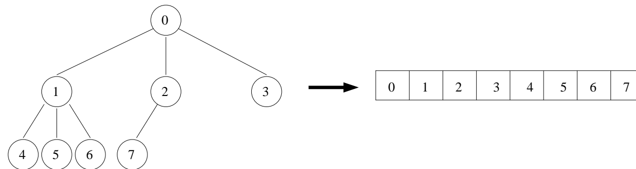


Figure 6: The array representation of a complete ternary tree. The array is densely packed, with the root coming first, then the elements at level 1, and then the elements at level 2.

shorter when $k > 2$, having $\lceil log_k n \rceil$ instead of $\lceil log_2 n \rceil$ levels; this reduces the number of iterations at the cost of increased iteration complexity. The optimum k is chosen as the value that minimizes the estimated running time in the performance model for a particular N. The k-ary tree is represented as a 1D array in the complete tree representation, similar to the Work-Depth description. The PRAM-on-chip implementation of this algorithm is presented in figure 3.b using the XMTC programming language.

We will consider the performance of the algorithm in the PRAM-On-Chip Execution Model in Section 4.4 after describing compiler optimizations.

4 Compiler Optimizations

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

4.1 Nested Parallel Sections

Quite a few PRAM algorithms can be expressed with greater clarity and conciseness when nested parallelism is allowed [Ble96]. For this reason, nesting parallel sections with arbitrary numbers of threads needs to be allowed in the PRAM-On-Chip Programming Model. However, hardware implementation of nesting is not free, and the programmer needs to be aware of the implementation overheads. In order to explain a key implementation problem we need to review the *hardware mechanism that allocates code 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()` command 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, and if not it will execute 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()` commands would require: (i) Upgrading this thread allocation mechanism. First, the number n representing the total number of threads will be repeatedly updated and broadcast to

the TCUs. (ii) Since a TCU gets just an integer result through the system `ps()` command, more information is needed to link this integer to a new thread that needs to execute. In addition, we need to facilitate a way for the parent (spawning) thread to forward initialization data to a child (spawned) thread.

In our prototype XMT PRAM-On-Chip 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). 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 prefix-sum 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 a `kspawn` command, see below) 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 vertices whose distance from starting vertex s is $i-1$ and outputs L_i. As noted in section 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, one per edge.

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 regarding write has request). 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 the data 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 PRAM-on-chip 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.

4.2 Clustering

The PRAM-On-Chip 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. The first option is to spawn fewer threads each doing more computation, while the second one is to run the shorter threads as is. Combining short threads into a longer thread is called clustering and offers several advantages: (a) 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) spawning fewer threads means reducing thread allocation overheads, i.e. the time required to start a new thread on a recently freed TCU; (c) each spawned thread (even those that are waiting for a TCU) usually takes up space in the system memory, to store the local data for the thread. If the code provides fewer threads than the hardware can support, there are fewer advantages if any to using fewer longer threads. Also, running fewer, longer threads

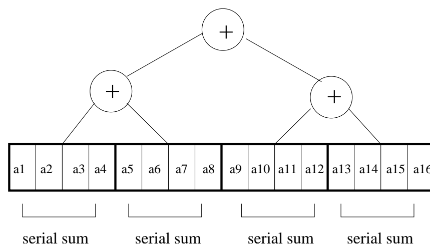


Figure 7: The sums algorithm with thread clustering.

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 embarrassingly parallel algorithm in which we spawn p threads that each serially sum $\frac{N}{p}$ elements and then the parallel summation algorithm is applied to the p sums. See figure 7.

With such switch to a serial algorithm, clustering is nothing more than a special case of the accelerating cascades technique [CV86]. 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.

When clustering is used an important question is to find the optimal crossover point between the slow (e.g., serial) and faster algorithms. As pointed out in the literature, accelerating cascades can be generalized to situations where more than two algorithms exist for the problem at hand.

A compiler should eventually be able to do the clustering automatically, though our current compiler does not yet does that. 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 complete $\frac{N}{c}$ threads in a serial manner. Sometimes an alternative serial algorithm can replace the N/c threads.

Clustering for single-spawn and k-spawn In the hardware, updates regarding the number of current virtual threads (either running or waiting) are broadcasted to TCUs as this number 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 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 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.

Using clustering, the number of running threads can be controlled. The best number of threads to run is not necessarily p, the number of TCUs. If several threads complete at the same time and, a gap in parallel hardware usage can occur, and it might take time to reach p running threads again. This can be avoided by having a threshold larger than p. One optimization is to run shorter threads as execution progresses toward completion of the parallel section. This will avoid a situation where all TCUs have already finished but one long thread is still running [NNTV03]. When using the method of clustering mentioned above using single-spawns and k-spawn, threads automatically become shorter as progress is made.

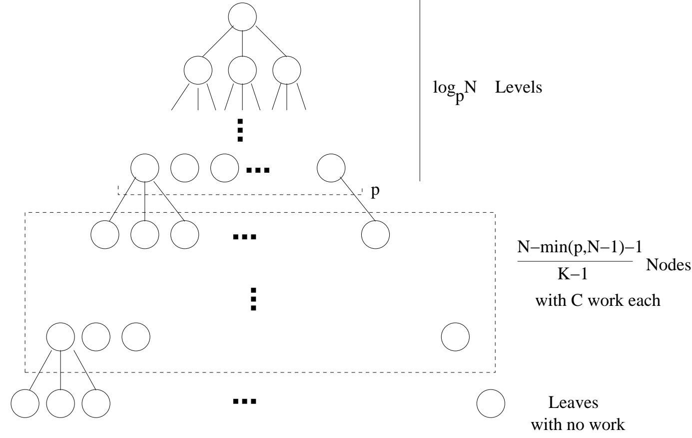


Figure 8: The $\log_p N$ levels of the tree closest to the root have $\frac{Work}{min(p,p_i)} = C$ (where C is the cost per node) because the parallel hardware is not saturated. The number of internal nodes at other levels is multiplied by C, then divided by p. The leaves do not need to be counted for work analysis.

4.3 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 [TKS+05, GBIV01]. Pai and Adve [PA01] 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.

4.4 Example: Analysis of Summation Algorithm

We analyze the Summation algorithm in the PRAM-On-Chip Execution Model. The computation advances by climbing (from leaves toward the root) a balanced k-ary tree. The algorithm has 2 RTMs for each level of the tree, one to read *sum* from a node's children (done in 1 RTM by prefetching all values) and one caused by the `join` command. As the basic algorithm fits an EREW PRAM, there is no queuing.

In addition to RTMs, the computation depth is $O(\log_k N)$ because a logarithmic depth tree is used. Counting constants factors on our own XMTC implementation yields the $(3k+9)\log_k N + 2k + 33$ portion of the execution depth term below. We have computed the computation per node to be $C = 3k+2$. The $\Sigma\frac{Work}{min(p,p_i)}$ term has a $\frac{2N}{p}$ component to copy data to the leaves of the tree and a $\frac{C*(N-min(p,N-1)-1)/(k-1)}{p} + C * \log_k(p)$ component, which is the cost to advance up 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), multiplying this by computation per node, and dividing by p. A level with less than p nodes has $\frac{Work}{min(p,p_i)} = \frac{(C\times p_i)}{p_i} = C$. See figure 8.

The overhead to start additional threads for oversaturated cases is computed similarly.

$$
\begin{aligned}
Execution\ Depth &= (2\log_k N + 1) \times \mathcal{R} + (3k+9)\log_k N + 2k + 33 \qquad (5)\\
Additional\ Work &= \frac{2N + (3k+2)(N - min(p, N-1) - 1)/(k-1)}{p} + \\
&\quad +(3k+2)\log_k p +
\end{aligned}
$$

$$+\left\lceil \frac{(N-min(p,N-1)-1)/(k-1)}{p} - log_k \frac{N}{p} \right\rceil \times \mathcal{R} \quad (6)$$

Clustering can be applied to the summation algorithm as summation allows switching to a serial algorithm when there is excess parallelism. The clustering algorithm starts off with an embarrassingly parallel algorithm and combines results at the end with the parallel summation algorithm, as follows:

```
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.  Each thread writes
   the sum it computed into an array B.
2. Call the parallel sums algorithm on the array B.
```

We now consider how clustering changes the execution time. c is the number of threads spawned in the embarrassingly parallel section of the algorithm. $SerSum(N)$ denotes the time for serial summation and $ParSum(N)$ the time of the parallel summation algorithm. The serial algorithm loops over N elements and, by using prefetching to always have the next value available before it is used, we derived that $SerSum(N) = 2N + 1 \times \mathcal{R}$. The execution time consists of first performing the serial algorithm on a set of $N - c$ elements (because it requires $N - c$ pairwise additions to sum c groups of N/c elements) divided evenly among p processors and then the parallel step. Namely,

$$Execution\ Time = SerSum(\frac{N-c}{p}) + ParSum(c) \quad (7)$$

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 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.

We found, not surprisingly, that for many (if not all) values $N \geq p$, the best c is 1024. Since clustering allows each thread to run a very efficient serial summation algorithm, the dry analysis implies the maximum possible clustering was the best (hence $c = 1024$).

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

5 Prefix-Sums

Prefix-sums 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-sum implementation approaches are presented and compared: The first algorithm considered is closely tied to the synchronous ("textbook") PRAM prefix-sums algorithm while the second one uses a no-busy-wait paradigm [Vis00]. The main purpose of the current section is to demonstrate designs of efficient PRAM-on-chip 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 [LF80], 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 9.

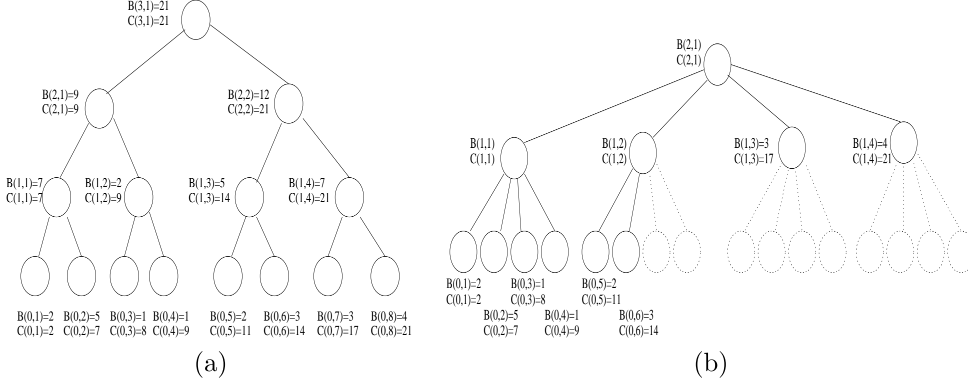


Figure 9: (a) PRAM prefix-sums algorithm on a binary tree and (b) PRAM prefix-sums algorithm on a k-ary tree (k=4).

5.1 Synchronous Prefix-Sums

The implementation of this algorithm in the PRAM-On-Chip Programming model is presented in figure 3.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 3.

The PRAM-On-Chip 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.

Analysis of Synchronous Prefix-Sums We analyze the algorithm in the PRAM-On-Chip Execution Model. The algorithm has 2 round-trips to memory for each level going up the tree. One is to read sum from a node's children, done in one RTM by prefetching all needed values at once. 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 a node's children. 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, We have analyzed an actual XMTC implementation and determined the constants 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 $\frac{C*(N-min(p,N-1)-1)/(k-1)}{p} + C * \log_k p$ 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 $Additional\ Work = \frac{(C \times p_i)}{p_i} = C$. 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:

$$Execution\ Depth = (4 \log_k N + 3) \times \mathcal{R} + (7k + 18) \log_k N + 2k + 39 \quad (8)$$

$$\begin{aligned} Additional\ Work = & \frac{3N + (7k+4)(N - min(p, N-1) - 1)/(k-1)}{p} + \\ & + (7k+4) \log_k p + \\ & \left\lceil \frac{(N - min(p, N-1) - 1)/(k-1)}{p} - log_k \frac{N}{p} \right\rceil \times 2\mathcal{R} \end{aligned} \quad (9)$$

5.2 No-Busy-Wait Prefix-Sums

A less-synchronous PRAM-On-Chip algorithm is presented. The Synchronous algorithm 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 [Vis00]. As before, we use a k-ary tree rather than binary trees.

The input and data structures are the same as previously, with the addition of the bi-dimensional array *gatekeeper* 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 PRAM-On-Chip 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 (k-1)-spawn to spawn 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 determined the constants involved.

Queuing is also a factor. In the current algorithm, up to k threads can perform the Prefix-sum operation at once on the same node and create a k queuing delay. The total QD on the critical path is $k\log_k N$.

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 $k\log_k N$ QD is added to this to make the $(11+9k)*\log_k N$ term. The $\Sigma\frac{Work}{min(p,p_i)}$ part of the complexity contains 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 $O(1)$ work that threads do 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:

$$\begin{aligned} Execution\ Depth &= (4\log_k N + 6)\times\mathcal{R} + (11+9k)*\log_k N + 2k + 55 \qquad (10) \\ Additional\ Work &= \frac{6 + 18N + (11+8k)(N - min(p,N-1)-1)/(k-1)}{p} + \\ &\quad +(11+8k)\log_k p + \end{aligned}$$

$$+\left\lceil \frac{(N - min(p, N-1) - 1)/(k-1)}{p} - log_k \frac{N}{p} \right\rceil \times 2\mathcal{R} \quad (11)$$

5.3 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 of
  step 1.
```

The serial summation algorithm is to iterate through the elements of the array and accumulate them in a *result* variable. The serial prefix-sum algorithm is the same as the summation algorithm except that, while iterating through the elements, the prefix-sum up to each element is stored in the result 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:

$$Execution\ Depth = SerSum(\frac{N-c}{p}) + SerPS(\frac{N-c}{p}) + ParPS(c) \quad (12)$$

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 $\mathcal{R} = 24$. We have checked that for many (if not all) values $N \geq p$, the best c is 1024. Clustering allows each thread to run a very efficient serial summation algorithm, and this implied the maximum possible clustering was the best ($c = 1024$). This is the case for both algorithms.

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. 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, as shown in figure 10.a. When $N < p$, clustering does not take effect, and the optimal value of k varies with N (for both algorithms).

5.4 Comparing Prefix-sums Algorithms

Using the performance model presented previously with these optimizations allows comparison of the programs in the PRAM-on-chip 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 10.b.

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

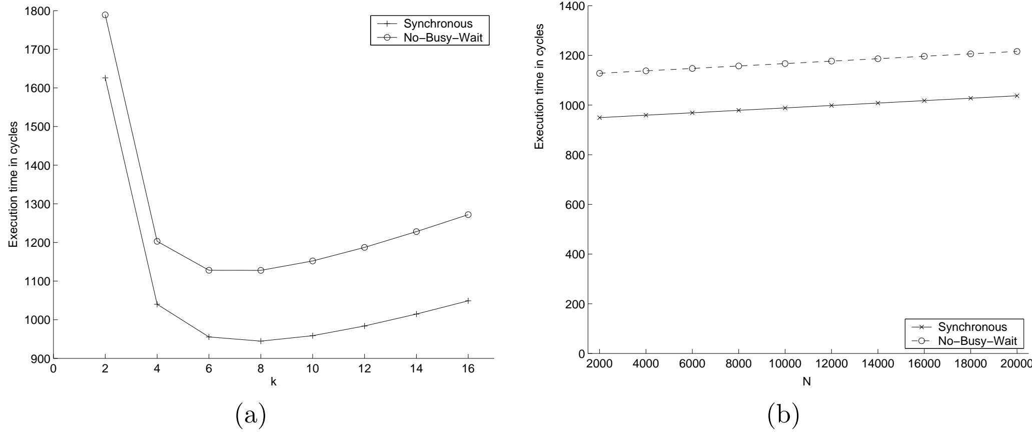


Figure 10: (a) Determining the optimum arity of the tree k for the two implementations of the Prefix-Sums algorithm for $N = 1024$. (b) Execution times for the two implementations of the k-ary Prefix-Sums algorithms. The optimum k is chosen for each case.

and more computation work due to the extra overhead of using gatekeepers and k-spawn. We note that in an actual PRAM-On-Chip system, prefix-sums would be likely to be included as a library routine that could be called by application developers.

6 Programming Parallel Breadth-First Search Algorithms

As noted earlier, Breadth-First Search (BFS) provides an interesting example for PRAM-On-Chip programming. We assume that the graph is provided using the incidence list representation, as pictured in figure 11.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 from that the full program is straightforward.

As described in section 2.2.1, 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 PRAM-on-chip 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.

6.1 Nested Spawn BFS

In a PRAM-on-chip programming model that supports nested parallel sections, the High-level PRAM-on-chip 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
```

A more detailed implementation of this algorithm using the XMTC programming language is included in figure 3.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 destination node, and the newly

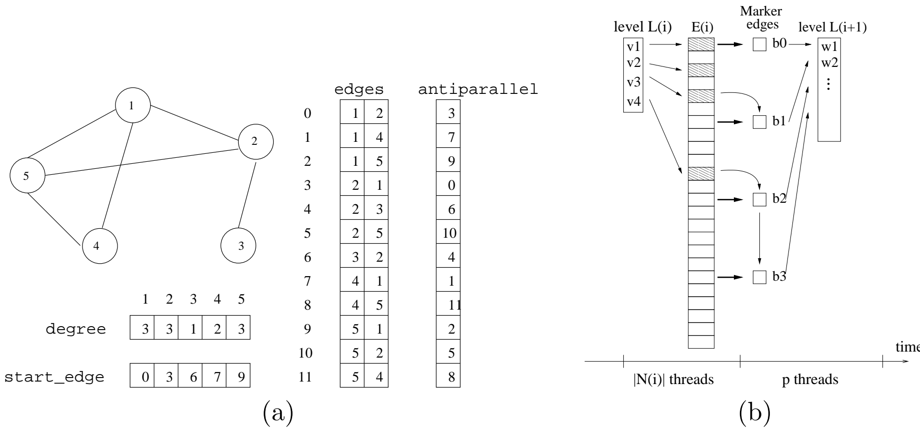


Figure 11: (a) The incidence list representation for a graph. (b) Execution of Flattened BFS algorithm. First allocate $E[i]$ to hold all edges adjacent to $level[i]$. Next, identify marker edges b_i, which give the first edge per each sub-array. Running one thread per sub-array, all edges are traversed to build $level[i+1]$.

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.

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 PRAM-on-chip framework.

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

6.2 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 11.

To identify the edges in each sub-array, it is sufficient to find the first (or *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 pointer jumping to identify all the marker edges are adjacent to v_j using at most $\log_2 p$ steps.

6.3 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 numbers T of threads in $\log_2 T$ steps. The algorithm spawns one thread for each vertex in the current level, and then ramps each thread up to $degree(vertex)$ 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 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 floor(s/(k+1)) edges
    Traverse (at most) k edges
```

7 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 PRAM-on-chip Execution Model as a framework to estimate running times.

7.1 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 in section 5. In our implementation, initializing all the TCUs with their proper subarrays uses 3 RTMs to initialize one edge entry per vertex, and then $4\log_2 p$ RTMs to do $\log_2 p$ rounds of pointer jumping and find all 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:

$$\begin{aligned} Execution\ Depth &= (4\log_k N(i) + 4\log_2 p + 14) \times \mathcal{R} + 38 + 10\log_2 p + 7|E(i)/p| + \\ &\quad +16N(i)/p + GQD + Computation\ Depth(Prefix\ sums) \qquad (13) \\ Additional\ Work &= \frac{7|E(i)| + 28N(i) + 15p + 17}{p} + \lceil \frac{N(i)-p}{p} \rceil \times \mathcal{R} + \\ &\quad +Additional\ Work(Prefix\ sums) \qquad (14) \end{aligned}$$

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

The second term of relation 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 to zero, apply to the relevant parallel sections clustering to run only p

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 $p' = \frac{|E(i)|}{\theta}$.

7.2 Single-spawn BFS

In this algorithm, one thread per each vertex v_i is started, and each of these threads then uses single-spawn to get $deg(v_i) - 1$ 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 round-trips to memory 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 round-trips to memory (as described in section 4.1), 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. Prefetching can be used here as well to reduce the number of roundtrips when multiple edges are traversed by one thread.

The cost of queuing at gatekeepers is represented by GQD. In our implementation, the additional computation cost 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 4.4 using a constant of $C = 19$ cycles work per node as implied by our implementation.

The performance is:

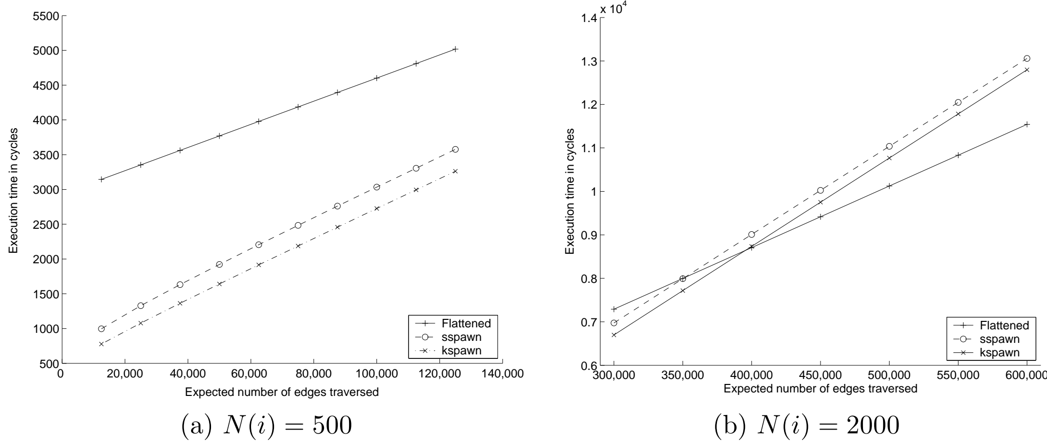
$$
\begin{aligned}
Execution\ Depth &= (7 + 2\log_2 d_{max})\mathcal{R} + (18 + 7\log_2 d_{max}) + GQD && (15)\\
Additional\ Work &= \frac{19(|E(i)| - min(p, |E(i)| - 1) - 1) + 2}{p} + \\
&\quad +19\log_2 |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} && (16)
\end{aligned}
$$

To avoid starting too many threads, the clustering technique presented in section 4.2 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.

7.3 k-spawn BFS

Similar to the case of the Single-spawn BFS algorithm, the k-spawn BFS algorithm uses a check of the number of virtual threads to determine whether the k-spawn instruction should continue to be used or if additional spawning is to be suspended.

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 4.1.



(a) $N(i) = 500$ (b) $N(i) = 2000$

Figure 12: Execution times when number of vertices is (a) $N(i) = 500$ and (b) $N(i) = 2000$. The optimal value for k was calculated for each dataset.

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:

$$Execution\ Depth = (7 + 2\log_k d_{max})\mathcal{R} + (5 + 4k)\log_k d_{max} + 15 + 4k + GQD \quad (17)$$

$$Additional\ Work = \frac{14|E(i)| + (4k+3)(|E(i)| - min(p, |E(i)| - 1) - 1)/(k-1)}{p} + +(4k+3)\log_k |E(i)| + \left\lceil \frac{|E(i)| - p}{p} \right\rceil \times \mathcal{R} \quad (18)$$

7.4 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 12. 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 11 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 12. 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 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. One idea for avoiding this 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). In the Flattened BFS algorithm however, the edges in a layer are evenly distributed among the running threads, and this issue does not appear.

8 Adaptive Bitonic Sorting

Bilardi and Nicolau's [BN89] 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. To implement the sorting, 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 PRAM-on-chip model, since it guarantees that no queuing delays will occur. On the other hand, PRAM-On-Chip 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. We will only consider the case where the problem size is $N = 2^n$; the general case is treated in [BN89]. 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 Nth 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 $\log_2 N$ recursive calls (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 [BN89], 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 determine direction of swapping
2.  swap root and spare values if necessary
3.  pl = root, pr = spare
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), 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 Pipelined 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, the performance of the pipelined version fell behind the simpler non-pipelined version. The main reason is the overheads required by the implementation of pipelining. Namely, some form of added synchronization, such as using a larger number of spawn blocks, is needed.

9 Shared Memory Sample Sort

The Sample Sort algorithm [HC83, RV87] 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 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 [BLM+91, RV87, DCSM96].

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 EREW PRAM algorithm that is better suited for a PRAM-On-Chip 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 that will partition the elements of A into p sets $bucket_i$, $0 \le i < p$ as follows: $bucket_0 = \{A[i] \mid A[i] < S'[0]\}$, $bucket_1 = \{A[i] \mid S'[0] < A[i] < S'[1]\}$, $\dots$, $bucket_{p-1} = \{A[i] \mid S'[p - 1] < A[i]\}$.

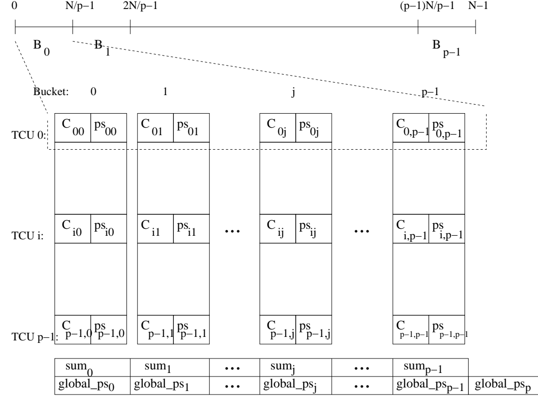


Figure 13: The helper matrix used in Shared Memory Sample Sort. c_{ij} stores the number of elements from B_i that fall in $bucket_j$. ps_{ij} represent the prefix-sums computed for each column, and $global_ps_{0..p}$ are the prefix-sums and the sum of $sum_{0..p-1}$.

Step 2. Consider the the input array A divided into p subarrays, $B_0 = A[0, \ldots, N/p - 1]$, $B_1 = A[N/p, \ldots, 2N/p - 1]$ etc. The ith 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 $bucket_j$. The c_{ij} make up the matrix C as in figure 13, (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 $S' = [100, 150, \ldots]$ 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 prefix-sums $ps[i, j]$ for each **column** of the matrix C. For example, $ps[0, j]$,$ps[1, j]$,...,$ps[p-1, j]$ are the prefix-sums of $c[0, j]$,$c[1, j]$,...,$c[p - 1, j]$.

Also compute the sum of column i, which is stored in sum_i. Compute the prefix sums of the $sum_1, \ldots, sum_p$ into $global_ps[0, \ldots, p - 1]$ 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 $Result[global_ps[i], \ldots, global_ps[i + 1] - 1]$.

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 PRAM-On-Chip 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 approach could be to scale down the architecture parameters by reducing the number of TCUs p, and estimate performance by extrapolating the results.

10 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. Next we discuss the so called *Matvec* problem of multiplying a sparse matrix by a

dense vector multiplication routine. Matvec is the kernel of many matrix computations. Parallel implementations of this routine have been used to evaluate the performance of other parallel architectures [FHKK05], [SG04].

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 14.

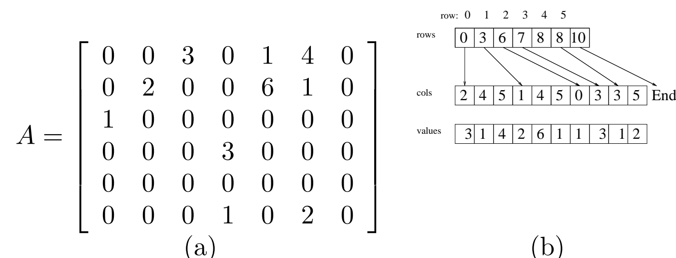


Figure 14: Compressed Sparse Row representation for a matrix. (a) A sparse matrix. (b) The "packed" CSR representation for matrix A.

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.

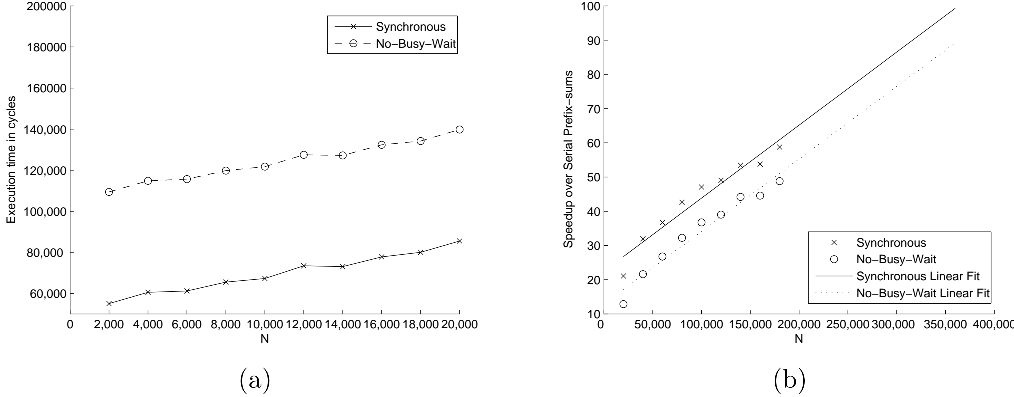
One factor that influences the performance of this algorithm on a PRAM-on-chip platform is the fact that the dense vector is read by all the parallel threads. On the XMT architecture, as TCUs request elements of the vector from memory, they are stored in the read-only caches at the cluster level, and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

The solution described above starts a thread for each row in the sparse matrix, and relies on the automatic load balancing and scheduling performed by the system to distribute the work between the available TCUs. However, if there are many rows in the matrix that have only few non-zero elements, it is not efficient to pay the overhead of starting one thread per each row. Alternatively, we can write an implementation which starts a smaller number of threads and attempts to better load-balance the work among them; a thread stops working when it has processed a certain minimum number of elements, but not before completing the row it is working on. Note that if there are rows that have a disproportionately large number of non-zero elements, it is nontrivial to split the work between threads since the results have to be added together; it is beyond the scope of this paper to present such an algorithm.

We have empirically compared the above two solutions by implementing the Matvec algorithms on the XMT PRAM-on-chip platform, and have found that the second solution, that better load-balances the work among the parallel hardware units, outperforms the embarrassingly parallel one for the range of inputs tested.

Given the first algorithm, it is a feasible, though perhaps somewhat challenging, task for a compiler to automatically generate the second, more efficient implementation. In general, if such a compiler can figure out that the information about the lengths of the threads will be known before the start of the parallel section, it could use this information to improve clustering. In the case of the first Matvec algorithm, 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.

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 the above two algorithms using XMTC in a course taught by Uzi Vishkin at the University of Maryland. Both courses were graduate courses. For the UCSB course this was the forth programming assignment and for the UMD course it was



(a) (b)

Figure 15: Execution times for k-ary prefix-sums ($k = 2$) implementations reported by the XMT cycle-accurate simulator. (a) Comparison of synchronous and no-busy-wait prefix-sums programs. (b) 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

the second. The main finding [HVGB05] was that XMTC programming required less than 50% of the time than for MPI programming.

11 Empirical Validation of the Performance Model

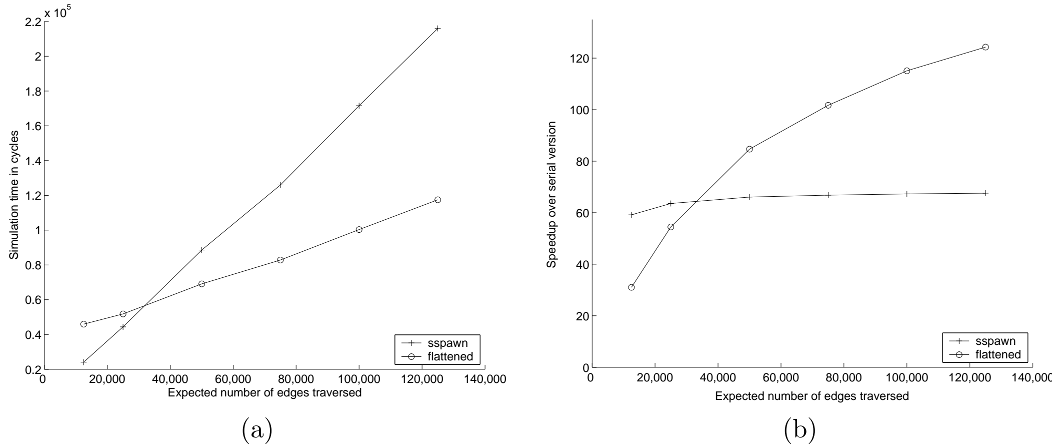
In this section we present a limited empirical attempt to validate the performance model developed in the previous sections.

We proceed by comparing the computed running times for the Prefix-Sums and the Breadth-First Search algorithms from sections 5 and 7 with cycle-counts of actual implementations which were ran on an XMT simulator. The simulation engine aims at being cycle-accurate and it was derived from a synthesizable gate-level description of the XMT architecture. A typical configuration includes 1024 TCUs grouped in 64 clusters and one Master TCU.

A gap between the simulations and the analytical model is to be expected. On one hand, the analytical model as presented makes some simplifying assumptions, such as counting each computation as one cycle and ignoring contention at the functional units and the interconnection network. 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 performance model; more specifically, there is no support for prefetching and thread clustering, and only limited broadcasting capabilities are included. Moreover, the sleep-wait mechanism proposed in section 4.1 is not fully implemented, making the single-spawning mechanism less efficient. All these factors cause the simulated cycle-counts to be higher than the ones computed by our formulas. However, we are mostly interested in comparing results for the same problem; a goal of the present work is to test programming approaches against the relative performance gain.

The results reported for k-ary prefix-sums ($k = 2$) by the XMT cycle-accurate simulator are shown in figure 15.a. The synchronous program outperforms the no-busy-wait program, verifying what had been found through the analytical model in figure 10. The execution times also increase linearly with N as was expected. One difference is the large gap that the simulator shows between the synchronous program and no-busy-wait execution times. This is explained by larger-than-anticipated overheads of the single-spawn instruction, which will be mitigated in future single-spawn implementations such as the sleep-wait mechanism previously described. Future use of k-spawn would also reduce the execution time. We also computed speedup results of parallel prefix-sums by running a serial implementation on the XMT simulator.

Figure 16.a presents the number of cycles reported by the XMT cycle-accurate simulator for the Single-Spawn and Flattened Breadth First Search XMTC implementations. The results show the execution times



(a) (b)

Figure 16: (a) Execution times for BFS implementations reported by the XMT cycle-accurate simulator when number of vertices is $N(i) = 500$ and (b) Speedups relative to a serial implementation for $N(i) = 500$

for only one iteration of the BFS algorithm, i.e. building BFS tree level $L(i+1)$ given level $L(i)$. The graphs are generated using the same procedure described in section 7.4, with the number of vertices in levels $L(i)$ and $L(i+1)$ set to 500. By varying the average degree per vertex M, we generated graphs with the expected number of edges traversed in the range $[12,500..125,000]$.

The results follow the same pattern as the ones presented in figure 12; similar to that figure, 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. Figure 16.b presents speedup results over a serial implementation of the BFS algorithm ran on the XMT simulator.

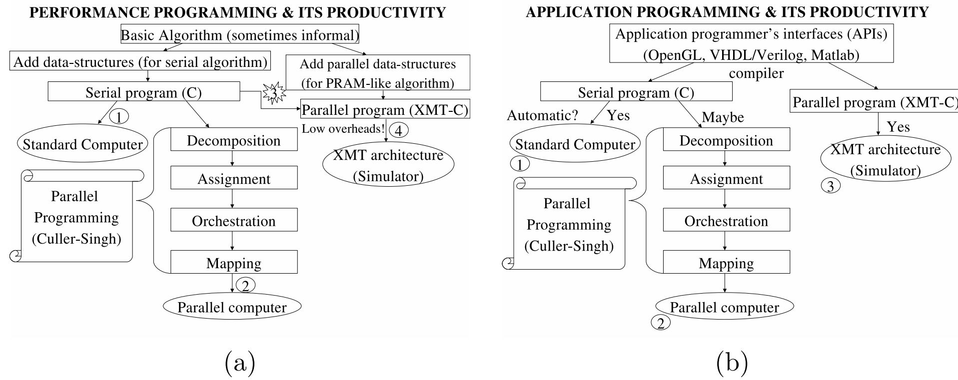
Comparing the outcome of our dry analysis with the experimental results serves a double purpose: on one hand, it makes us aware of the limitations of the analytical model and allows further refinements to more closely match the architecture; at the same time, the benefit of new features can be easily evaluated using the analytical model and then confirmed in the simulator, before committing to a much more costly update to the hardware architecture.

Further information about the XMTC compiler and the cycle-accurate simulator can be found in [BV06]. 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`.

12 Conclusion

The programming assignments in a one semester parallel algorithms class taught recently at the University of Maryland included parallel MATVEC, general deterministic (Bitonic) sort, breadth first search on graphs, and sample sort. This fact provides a powerful demonstration that the PRAM theory coupled with PRAM-On-Chip programming are on par with serial algorithms and programming. A first class on serial algorithms and serial programming typically does not require more demanding programming assignments. 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 PRAM-On-Chip computer system to allow such teaching elsewhere.

It is also interesting to compare the PRAM-On-Chip 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 PRAM-on-chip 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



(a) (b)

Figure 17: (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).

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 17 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 [CS99], with the PRAM-on-chip 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 needs to be done, for the PRAM-On-Chip 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

[AALT95] 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.

[ACK87] 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.

[AG94] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, 1994.

[AU94] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. W. H. Freeman & Co., New York, NY, USA, 1994.

[Baa88] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1988.

[BCF05] David A. Bader, Guojing Cong, and John 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.

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[BLM+91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 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.

[BN89] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, 1989.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[BV06] Aydin O. Balkan and Uzi 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.

[CFS99] L. Carter, J. Feo, and A. Snavely. Performance and programming experience on the tera mta. In *Proceedings SIAM Conference on Parallel Processing*, 1999.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[CV86] Richard Cole and Uzi 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.

[DCSM96] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. Fast parallel sorting under logp: Experience with the cm-5. *IEEE Trans. Parallel Distrib. Syst.*, 7(8):791–805, 1996.

[DKP02] Roman Dementiev, Michael Klein, and Wolfgang 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.

[DV00] Shlomit Dascal and Uzi 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.

[EG88] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *Annual review of computer science: vol. 3, 1988*, pages 233–283, 1988.

[FHKK05] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[GBIV01] 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.

[GGK+82] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc 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.

[GV06] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extermely 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.

[HC83] 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.

[HJ99] David R. Helman and Joseph JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, pages 37–56, London, UK, 1999. Springer-Verlag.

[HVGB05] Lorin Hochstein, Uzi Vishkin, John Gilbert, and Victor Basili. An empirical study to compare the productivity of two parallel programming models. Preprint, 2005.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[KKT00] J. Keller, C. W. Kessler, and J. L. Traff. *Practical PRAM Proramming*. Wiley, New York, NY, USA, 2000.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 869–941, 1990.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[Man89] U. Manber. *Introduction to Algorithms - A Creative Approach*. Addison Wesley, 1989.

[NNTV03] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi 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.

[PA01] Vijay S. Pai and Sarita 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.

[RV87] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.

[SCB+98] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian 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.

[Sch80] Jacob T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, 1980.

[SG04] Viral Shah and John R. Gilbert. Sparse matrices in matlab*p: Design and implementation. In *HiPC*, pages 144–155, 2004.

[SV82] Yossi Shiloach and Uzi Vishkin. An $o(n^2 logn)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.

[TKS+05] Xinmin Tian, Rakesh Krishnaiyer, Hideki Saito, Milind Girkar, and Wei 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.

[VDBN98] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph 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.

[Vis00] Uzi 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.

[Vis02] Uzi 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.

APPENDIX. The XMTC Code is submitted in support of the paper.

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; //FOR DEBUGGING ONLY, REMOVE LATER
        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 = l;
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; //FOR DEBUGGING ONLY, REMOVE LATER
```

```
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 = l;
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 < l) {
```

```
        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; //FOR DEBUGGING ONLY, REMOVE LATER
        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;
}

/*
 * 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
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[$]; // 1 RTM
                isLeaf[$ + sbp1] = 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_i) =
     * ((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. UNITIALIZED 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 = Ei/p times.
         * We assume Ni 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=Ei/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 Ni + 4 |Ei|/p + 11 + LSRTM of PSUMS
    *   W PRE  : RTM Time = 4 log_k Ni + 4  + 11 + 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[$] > 0) { // 1 computation
      // this is executed for each child thread spawned
       sspawn(newTID) { // 1 RTM
         /*
          * writing initialization data for child threads.
          * children will wait till this data is commited
          */
          nForks[newTID] = (nForks[$]+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 commited
              */

             slice = nForks[$] / k;
             nForks[$] = nForks[$] - slice; // substract 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];
             }
              /*
               * loop is executed k times.
```

```
            * Each iterration:
            *   0 RTM
            *   4 computation
            */
    }
    /* For each k child threads:
     *   1 RTM
     *   2+4*k computation
     */
} // done with forking
```