Practical Parallel Pattern Matching on Shared Memory Multiprocessors

Maria Hybinette
and
Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

(ingrid,schwan)@cc.gatech.edu

Submitted to Concurrency-Practice and Experience

June 3, 2004

Summary

Theoretically optimal parallel pattern matching algorithms assume models of computation that do not accurately represent parallel hardware. The specific issue investigated in this paper is whether algorithms evaluated in performance and compared with the PRAM model exhibit similar relative performance on physical machines. Our conclusion is that this is not the case, because such performance evaluations do not take into account runtime overhead due to remote memory access, synchronization or inter-task communication costs. To validate this conclusion, three parallel pattern matching algorithms are evaluated on a 64 processor KSR2 supercomputer, a 12 processor SGI Power Challenge and a 16 processor SGI Origin 2000. One algorithm (Vishkin's) is theoretically optimal and asymptotically faster when implemented on a PRAM architecture; the others (the Boyer-Moore and the Knuth-Morris-Pratt algorithms) are serial algorithms parallelized by domain decomposition. Processing time is measured on 1 to 40 processors for searches of primate DNA sequence data 1 to 128 megabytes long, using long and short patterns (2 bytes to 10 megabytes). Performance is also evaluated on English and random texts. In all cases, the parallelized serial algorithms out-perform the "optimal" algorithm of Vishkin.

Index Terms– Experimental algorithmic, parallel models, multiprocessors, pattern matching algorithms.

Practical Parallel Pattern Matching on Shared Memory Multiprocessors

Submitted to Concurrency-Practice and Experience

Summary

Theoretically optimal parallel pattern matching algorithms assume models of computation that do not accurately represent parallel hardware. The specific issue investigated in this paper is whether algorithms evaluated in performance and compared with the PRAM model exhibit similar relative performance on physical machines. Our conclusion is that this is not the case, because such performance evaluations do not take into account runtime overhead due to remote memory access, synchronization or inter-task communication costs. To validate this conclusion, three parallel pattern matching algorithms are evaluated on a 64 processor KSR2 supercomputer, a 12 processor SGI Power Challenge and a 16 processor SGI Origin 2000. One algorithm (Vishkin's) is theoretically optimal and asymptotically faster when implemented on a PRAM architecture; the others (the Boyer-Moore and the Knuth-Morris-Pratt algorithms) are serial algorithms parallelized by domain decomposition. Processing time is measured on 1 to 40 processors for searches of primate DNA sequence data 1 to 128 megabytes long, using long and short patterns (2 bytes to 10 megabytes). Performance is also evaluated on English and random texts. In all cases, the parallelized serial algorithms out-perform the "optimal" algorithm of Vishkin.

Index Terms– Experimental algorithmic, parallel models, multiprocessors, pattern matching algorithms.

1 Introduction

The Parallel Random Access Machine (PRAM) is a standard theoretical model of computation used to predict the performance of parallel applications [12]. The PRAM comprises an unbounded number of processors accessing an infinite space of shared memory without incurring communication costs for these accesses. Weaknesses of the PRAM model have been pointed out by numerous researchers who in turn have proposed new computational models to account for specific inaccuracies [1, 10, 11, 13, 18, 20, 30, 33]. Although the literature is expanding towards more realistic models, the PRAM model remains in frequent use because of its simplicity and generality (e.g. [8, 16, 21]).

The memory access time not accounted for in the PRAM leads to unpredictable behavior of applications, especially when memory access time is large with respect to computation time. PRAM model argue that smart hardware or compiler pre-fetching [25, 35] can hide actual memory access latencies. Current hardware or compilers do not have such capabilities and, unfortunately, the trend toward faster CPUs and larger memory hierarchies (register, sub-cache, cache, and various levels of remote memories) suggests that problems with the PRAM model will worsen rather than improve.

This paper experimentally demonstrates the difficulty of *comparing* algorithms in the PRAM model. We compare a theoretically optimal and asymptotically faster parallel algorithm with two serial algorithms parallelized by domain decomposition. Specifically, we compare an optimal PRAM algorithm solving a pattern matching problem by Vishkin [34] with two serial pattern matching algorithms, one by Boyer and Moore [5] and the other by Knuth, Morris and Pratt [22]. The results of our work demonstrate the problem of comparing parallel algorithms using simplified theoretical models. Even though Vishkin's algorithm is optimal and asymptotically faster under PRAM, the parallelized serial algorithms outperform it on real hardware by significant margins. Performance is measured while varying several parameters including the number of processors, text length, pattern length and text source differing in the size of the alphabet (DNA, random text, and various English texts, with alphabet sizes of 4, 95 and 75-84 respectively). In all cases, the parallelized serial algorithms out-perform Vishkin's. Figure 1 is representative of our results. The graph shows performance of the three algorithms searching for a pattern 16 bytes long in an English text of size 128 megabytes. The significant result of this research is that use of the PRAM model for performance prediction leads to incorrect statements about the suitability of various parallel algorithms running on realistic and existing parallel architectures. This is particularly the case for fine-grained parallel

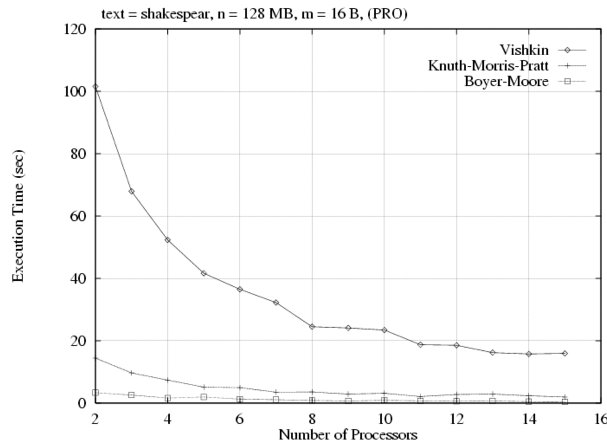

Figure 1: Performance of the Vishkin algorithm and the parallelized Boyer-Moore and Knuth-Morris-Pratt algorithms on a 16 processor SGI Origin 2000. Lower values indicate better performance.

algorithms where the PRAM model predictions currently do not meet and are not likely to meet in the future, the measured algorithmic performance. Realistic predictions must account for delays in memory access, synchronization and communication costs.

There exist other computational models such as the Bulk Synchronous Parallel (BSP) [33], Block Distributed Memory (BDM) [20], and LogP [9] that account for non-uniform memory access and synchronization costs. These models are motivated by the deficiency of the PRAM, as is our research. Our paper complements the theoretical work underlying these new models by exposing thoroughly and experimentally the practical reasons underlying the issues with PRAM.

Experimental evaluations of algorithms provide useful insights. In particular, we clearly show experimentally, the reasons why it is difficult to predict actual execution times on parallel machines for algorithms that differ asymptotically on the PRAM model. While it is well-known that PRAM can be improved, there is also intellectual value experimentally examining the reasons underlying the need for such improvements.

This paper is organized as follows. The following section describes the PRAM model and the experimental hardware platforms used in this research. The task examined, namely pattern matching, is discussed in Section 3. Section 4 describes Vishkin's parallel pattern matching algorithm and the serial Boyer-Moore and Knuth-Morris-Pratt algorithms. The parallel implementations of the Knuth-Morris-Pratt, Boyer-Moore and Vishkin algorithms and their performance on real machines are described in Sections 5 and 6. An improved implementation of the Vishkin algorithm and its performance are discussed in Section 7. Section 8 discusses results indicating that the Vishkin algorithm may never achieve "optimal" performance in practice even if the number of processors are unlimited as the PRAM architecture assumes. We conclude with a

discussion of the implications on our results and future directions in computer hardware development.

2 Platforms

2.1 The PRAM Architecture

The PRAM is a shared memory model where processors work synchronously, executing the same program. The model assumes an unbounded number of processors that communicate through shared memory or global random access memory. The memory is of unbounded size and is uniformly accessible by all processors. This architecture is illustrated in Figure 2. Each memory cell can be independently accessed

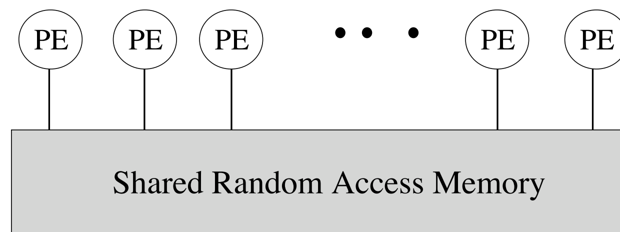

Figure 2: The PRAM architecture, unbounded number of processors.

without cost, even when there is contention between neighboring memory cells. Since the communication is “free”, memory bandwidth is unlimited and any memory access take unit time.

“PRAM” is often preceded by a prefix to denote assumptions concerning the handling of simultaneous memory accesses. For example, the concurrent-read concurrent-write (CRCW) PRAM allows both concurrent read and concurrent writes to the same memory location, and the concurrent-read exclusive-write CREW-PRAM allows simultaneous reading of the same memory location but does not allow simultaneous writing.

2.2 The Ring Based Architecture

The first parallel architecture used for this research is an example of a well-balanced machine: a 64 processor Kendall Square Research (KSR2) supercomputer. The KSR2 is a cache-only-memory architecture (COMA). The processors are 64-bit two-way set-associative super-scalar RISC CPUs operating at 40 MHz each. Peak performance is 40 M-flops per node, a speed that is well matched by the machine’s bus architecture described next.

The KSR is classified as a NUMA (Non-Uniform Memory Access) system, where the memory access times are non-uniform depending upon whether the location addressed is in local memory or in various

levels of remote memory. The "bus" topology of the KSR is a hierarchy of rings where each ring has 34 slots. 32 of the slots are used for connecting processors and the remaining two are used for linking to other rings. The 32 processors form a ring operating at 1 GB/sec. Interconnection bandwidth within a ring scales linearly, since every ring slot may contain a transaction [4], resulting in a ratio of CPU to bus capacity that is far better than that of other parallel architectures discussed in the literature. This is especially true for the increasingly popular networked parallel machines now employed widely [26]. The KSR2 was intended to support 5,000 processors although the ring design supports an arbitrary number of levels.

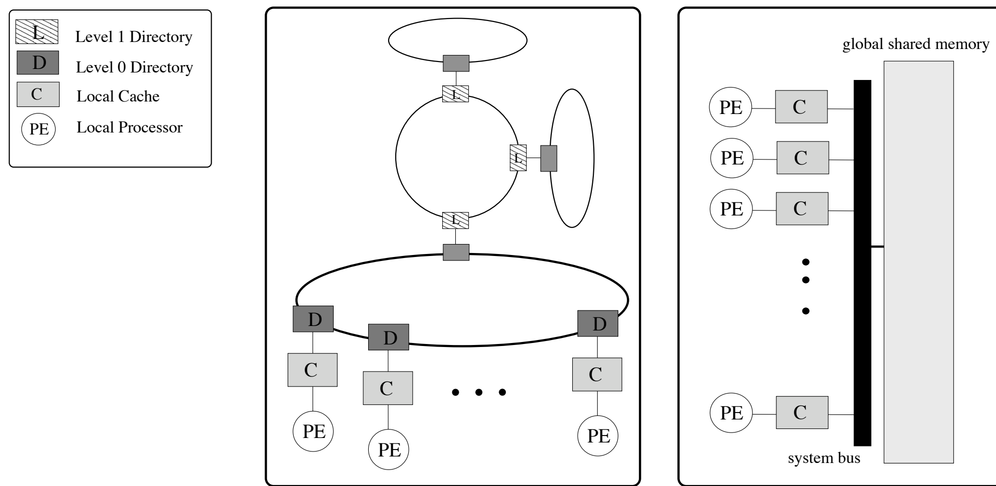

Figure 3: The ring-based architecture on the left and the bus-based architecture on the right.

The KSR2 stores data in units of pages and sub-pages. Pages are of size 16 K-bytes, divided into 128 sub-pages of 128 bytes. The unit of allocation in local caches is a page. The unit of transfer and sharing between local caches is a subpage. Each local cache has room for 2,048 pages or a total of 32 M-bytes. A sub-cache has room for 0.25 MB of instructions and 0.25 MB for data. When a processor references an address not found in its local cache, the memory manager first makes room for it by allocating a page. The contents of the newly allocated page are filled as needed, a subpage at a time.

Memory page routing is controlled on the KSR using the ALLCACHE hardware engine. The ALLCACHE provides a single shared address space for all processors throughout the ring architecture, where 40-bit addresses are supported.

Via hardware migration and replication, the ALLCACHE engine provides a sequentially consistent memory model. With sequential consistency, the result of any execution is the same as if the operations of all processors where executed in some sequential order and the operations of each individual processor

appear in this sequence in the order specified by its program [23]. Memory coherence is maintained by updating all cells when a processor writes to an address. Memory access (which occurs in sub-pages of 128 bytes) latencies are approximately 2, 18, 126, and 600 cycles for local sub-cache, local cache, AG:0 (consisting of up to 32 CPUs) and AG:1 (consisting of up to 1088 CPUs), respectively. In comparison to more recent architectures, the KSR's memory scheme offers more favorable ratios of memory access times to CPU cycle times. Specifically, the latencies for the SGI Origin 2000, another NUMA architecture discussed below, are 11, 61 and 138 cycles for the L2 hardware cache, local memory cache shared by two CPUs and a remote memory of up to 32 nodes, respectively [24, 29].

The KSR differs from the PRAM in that memory access times are non-uniform depending on the location of data. By treating all memory as a cache (COMA), hardware supports some measure of latency hiding in memory access, as data is automatically moved to a processor when needed. This functionality helps bridge the gap between PRAM and the KSR, but nevertheless the KSR machine has to physically move or copy data to realize memory accesses.

2.3 The Bus Based Architectures

The second class of parallel architectures used in our work represents the large number of bus-based parallel machines employed in practice: an SGI Power Challenge. Our SGI Power Challenge has twelve 75 MHz MIPS R8000 processors. In this machine, the first level instruction and data caches are of size 16 KB each. The unified secondary cache has a size of 4 MB. The main memory size of the Power Challenge is 3 GB. The latencies for the SGI Power Challenge are 5, 53 and 80 cycles for local sub-cache, local memory and remote memory (where the bus supports the use of 12 processors simultaneously), respectively [24, 28, 29]. The processors on the Power Challenge communicate via a fast shared-bus interconnect. The bus has a bandwidth of 1.2 gigabytes (GB) per second with a 256-bit wide data bus and a separate 40-bit wide address bus that can access up to one terabyte (TB) of physical memory. The bus provides high-bandwidth, low-latency, cache-coherent communication between processors, memory, and I/O. As a result the processors share a single pool of memory where memory access cost is identical for all processors. Cache coherency is maintained by snooping on the bus. The significant aspect studying bus-based architectures is that they are classified as uniform memory access (UMA) systems.

Bus-based architectures differ from the PRAM model's assumptions in that they have finite bandwidth, and only a finite number of transactions can be performed per cycle. This limits the scalability of these

architectures. In contrast, the PRAM model assumes that it can support an infinite number of processors without increased memory access costs.

2.4 The Hub Based Architecture

The third class of architectures considered in this research is represented by an SGI Origin 2000 with 16 195 MHz MIPS R10,000 processors. Its first level instruction and data caches are of sizes 32KB each. The unified secondary cache is of size 4 MB. The main memory size of the Origin is 4 GB. The Origin, like the KSR, is classified as a NUMA machine. The architecture is hub-based with up to two processors per

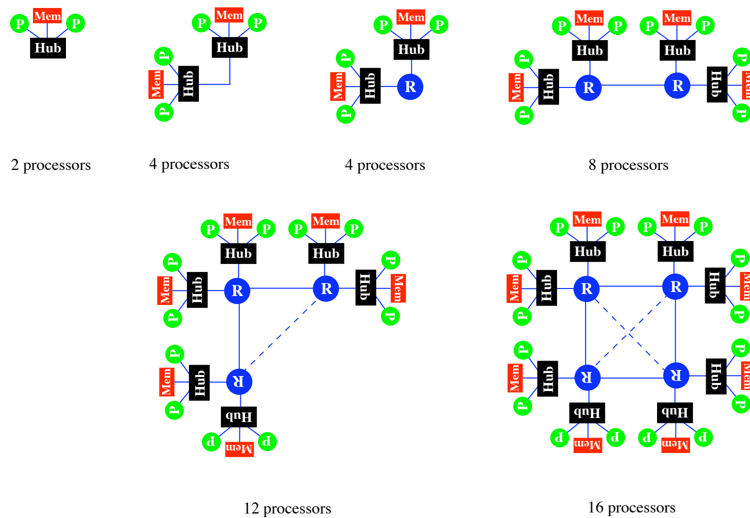

Figure 4: The hub based architecture of the SGI's Origin 2000.

hub. The hub contains memory that can be shared between the two processors. Figure 4 shows SGI Origin 2000 topologies for 2-16 processors. The hub manages memory accesses and I/O. Larger systems are built by connecting hubs to each other or to a router. A router has six ports that can connect to additional hubs or routers (a router is shown as an R in Figure 4). The advantage of adding a router to a hub rather than an additional hub is that larger systems can be built since two hubs connected to each other allow for a maximum system of 4 processors; the disadvantage is that the use of routers leads to higher memory access costs. Systems are constructed by increasing the dimensionality of the router configuration and adding up to two hubs with each additional router. Systems with any number of nodes may be constructed by leaving off some corners of the n-dimensional hypercube (see Figure 5).

The Origin uses a directory-based cache coherence scheme. Memory is organized in cache lines of 128 bytes. The directory cache coherency mechanism uses dedicated circuitry in the hub to manage memory.

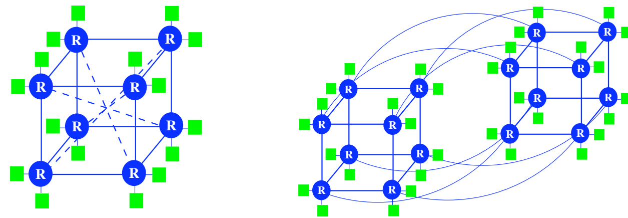

Figure 5: Larger configurations of the SGI's Origin 2000, 32 and 64 processors.

The latest copy is retrieved without waiting to write back to memory. When a node requests a cache line, its hub initiates the memory fetch of the whole line from the node that contains that line.

The Origin 2000 differs from the PRAM model in that it is NUMA (as with the KSR multiprocessor). Therefore, locality of memory access must be taken into consideration. Latency hiding is accomplished by caching and pre-fetching data that are not cache-resident. Furthermore, the topology of the hypercube router configuration implies that memory accesses pass through at most $n + 1$ routers, where n is the dimension of the hypercube. The Origin requires more cycles than the Power Challenge to access various levels of memory (Origin 2000 requires 11, 61 and 138 cycles respectively for accesses to the L2 cache, local memory shared by two CPUs and remote memory of up to 32 nodes, respectively while the Power Challenge requires 5, 53 and 80 cycles respectively to access cache, main memory and remote caches; this implies a degradation in the ratio memory access time to CPU speed for this machine compared to the less scalable Power Challenge.

2.5 Discussion of Platforms

It is not feasible to realize the PRAM model's assumptions of both scalability and uniform memory access costs in one architecture. Existing hardware can provide either of these capabilities, but one is typically provided at the expense of the other.

Bus-based architectures for instance, can provide uniform memory access costs for a given number of processors, where memory accesses have the same latencies regardless of their target memory locations. Unfortunately, bus-based systems fail to emulate the PRAM in three ways. First, they do not scale; adding processors is limited by bandwidth constraints on the bus. Second, there are limits on the number of processors that can simultaneously access the same location or even the same set of locations (i.e. memory bank) in main memory, due to bounds on memory concurrency. Third, synchronization instructions have substantially higher overheads than memory accesses, leading to additional scalability problems. All of these behaviors violate the assumptions of the PRAM model.

NUMA architectures are subject to bus-based machines' problems with concurrent memory access and with the cost of synchronization instructions. However, they provide improved scalability by sacrificing uniformity in memory access costs. In these systems, main memory is distributed among the processors. When one processor must access a remote memory cell, the request is transmitted to the remote node. The number of hops required depends on the number of processors in the system. Again, this violates a key assumption of the PRAM model.

It is not likely that industry will devise a system providing both scalability and uniform memory access. The market is moving towards scalable multi-layered memory architectures with low average memory access costs due to multiple layers of cache and local memory, but with higher costs for more distant memory accesses. Market trends also indicate a growing disparity between processor speeds and memory access times (see Figure 6). Processors are getting faster, while memory speeds are remaining about the same.

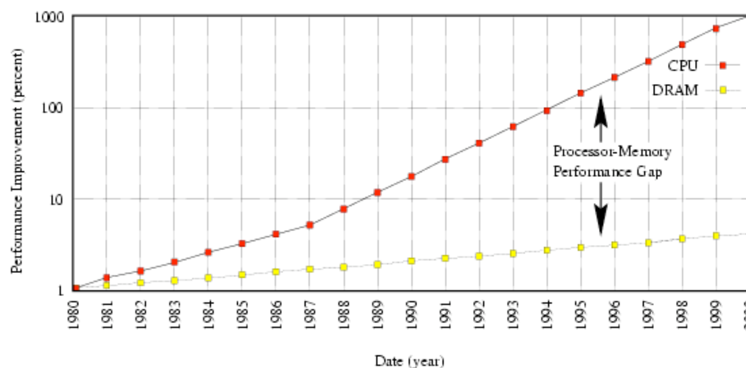

Figure 6: Percent improvement in performance since 1980 for microprocessor and DRAM access time. The processor-memory performance gap is increasing: Microprocessor performance improves at a rate of 60% per year and the access time to DRAM improves at a slower rate of less than 10% per year [19].

This trend is also evident in the architectures studied in this paper. For example, Figure 7 shows that the older NUMA based KSR is a better balanced machine than the more recent NUMA based SGI Origin 2000. This trend will move real machines even further from the PRAM model.

Finally, for all parallel hardware, the cost of a memory access varies significantly depending on the machine architecture, on how recently the memory cell was accessed by the current processor and on how recently it was written by another processor. PRAM does not capture such performance effects derived from the locality of data. Furthermore, for problems like pattern matching (discussed in detail in the next section) where the input size (text, and pattern) is large, the PRAM allows the number of concurrently executing tasks to grow as a function of the size of the input. This leads to inaccurate performance estimates because the PRAM does not account for the overhead of context switching between tasks.

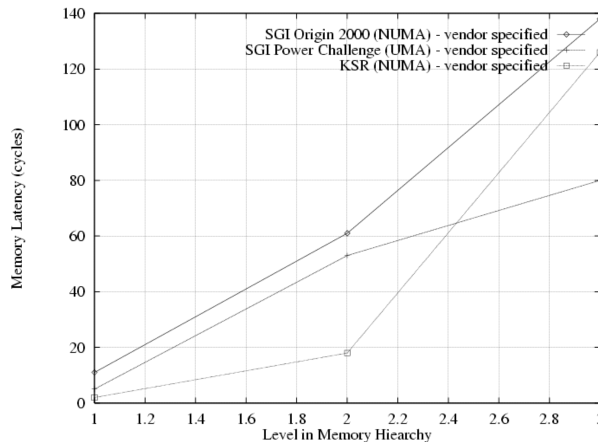

Figure 7: Memory access cost at various levels of memory for the SGI Origin 2000, the KSR and the SGI Power Challenge.

3 Parallel Pattern Matching

The experimental aspect of this work addresses *pattern matching* algorithms, where all occurrences of a given **pattern** are found in a specified **text**. Pattern matching algorithms are important in building information retrieval systems, managing large databases or exploring and searching the Internet. The theoretical performance of pattern matching algorithms is described in terms of the length of the text to be searched, n, and the length of the pattern to be found, m. Performance may also depend on the alphabet size c, which is the number of different symbols used in the text and pattern. Various sequential methods exist for pattern matching: dynamic programming [31], suffix trees [32] and automaton-based algorithms [2, 5, 22]. This study focuses on non-periodic [1] patterns.

Optimal parallel pattern matching algorithms have been developed by Galil [15] and by Vishkin [34], where the former provides for a constant alphabet size and the latter assumes an arbitrary alphabet size. For a parallel algorithm, "optimal" means that the work (the total number of operations) of the parallel algorithm is within a constant factor of the work of the best known sequential algorithm solving the same problem [2]. These algorithms are developed for the PRAM model. Vishkin's optimal parallel algorithm runs in the general case in $O(\log m)$ on a CRCW-PRAM having $n/\log m$ processors and in $O(\log^2 n)$ on a CREW-PRAM having $n/\log^2 n$ processors. Thus, for both types of PRAM, more than $350,000$ processors would be required to search for an eight byte pattern in a 10 megabyte text. When there are fewer processors available, the algorithm runs in $O(n/p)$ time for both the CRCW-PRAM and the CREW-PRAM.

[1] A string s of length l is periodic if the length of its period is at most half its length l. Here, a string u is a period of a string w if w is a prefix of u^k for an integer k, or equivalently if w is a prefix of uw [15]

[2] i.e. a parallel algorithm is optimal if it takes $O(S(n)/p)$, where p is number of processors, n is the problem size, and $S(n)$ is the worse case performance of the best known sequential algorithm solving the same problem

For purposes of comparison, we parallelize the Boyer-Moore serial algorithm by domain decomposition, motivated by the fact that this algorithm prevails in practice. The algorithm is attractive despite its worst-case performance of $O(mn)$ since on average it performs sub-linearly. Another serial algorithm by Knuth-Morris-Pratt is chosen because its worst-case performance is linear at $O(n + m)$. Parallelization of a serial algorithm by domain decomposition is attractive because certain memory reference costs can be minimized in-part, because each processor is solely responsible for processing a specific portion of the domain without sharing data with other processors. This also avoids false sharing, where different processors accessing different memory units contend for the same memory page. In contrast, Vishkin's algorithm depends on the locality of data and this implicates its theoretical performance. Specifically, our results show that the Boyer-Moore and the Knuth-Morris-Pratt algorithms outperform the Vishkin algorithm on all platforms.

4 Three Algorithms for Pattern Matching

4.1 Vishkin: A Theoretically Optimal Parallel Algorithm

Vishkin's algorithm has three steps:

Input:	Two one-dimensional arrays: T [1..n] – the text, P [1..m] – the pattern.
Output:	All occurrences of the pattern in the text.
Step I:	Pre-process the pattern to generate the witness array.
Step II:	Eliminate candidates by duel.
Step III:	Test each surviving candidate for a match.

To explain Vishkin's algorithm, we must first introduce the concept of a "duel" and the representation of the "witness" array. The basic idea is that for any two prospective candidates in the text fewer than m characters apart, only one may survive a "duel" (remain a possible match candidate). Only one such match can exist[3]. For example, if j_1 and j_2 are two positions in *text*, and $|j_1 - j_2| < m$, then it is impossible for (an aperiodic) $pattern$ to match the text at both locations (see Figure 8). To resolve a duel between positions j_1 and j_2 of *text*, the algorithm refers to a **witness** array ($witness$) which is constructed when the pattern is pre-processed. $witness[i]$ indicates the first position of a mismatch when the pattern is compared with

[3]This is true for non-periodic patterns. In a non-periodic pattern, no suffix of the pattern matches any prefix of the pattern. Periodic patterns require an extension to the basic Vishkin algorithm which is not implemented here.

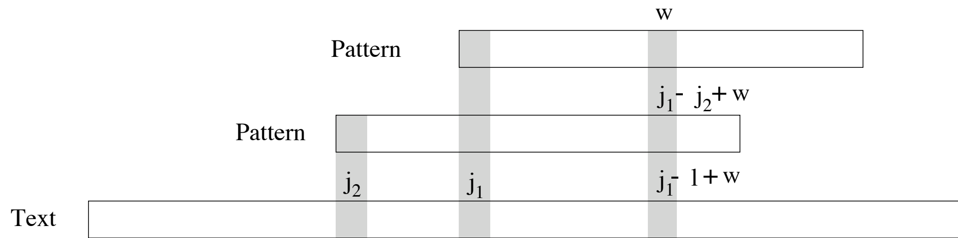

Figure 8: Candidate matches j_1 and j_2 spaced less than m apart cannot both have a match with pattern of size m.

itself at position i ($witness[1]$ is always 0, because the pattern always matches itself). To illustrate this idea consider the pattern `daddadad` and shift another copy of the pattern above it:

								no mismatches
d	a	d	d	a	d	a	d	$witness[1] = 0$
d	a	d	d	a	d	a	d	
1								shift $= 1$

For this pattern, $witness[1]$ is 0 because the pattern matches itself. The shift position for the above case is 1 and the shift is also the index of the witness array. When we shift the above pattern one position further to the left (a shift position of 2), we find that $witness[2]$ is 1 since the first character "`d`" mismatches the second character "`a`". This situation is shown below:

	1								mismatch at 1
	d	a	d	d	a	d	a	d	$witness[2] = 1$
d	a	d	d	a	d	a	d		
	2								shift $= 2$

When a match is attempted at a shift of 3, a mismatch is first found at position 2 of the pattern above, so $witness[3]$ is 2. This is illustrated below:

		1	2							mismatch at 2
		d	a	d	d	a	d	a	d	$witness[3] = 2$
d	a	d	d	a	d	a	d			
		3								shift $= 3$

Continuing shifting the above pattern to compute the witness array in the same manner, we have $witness[4]$ is 4 (see below) and $witness[5]$ is 1:

			1	2	3	4					mismatch at 4
			d	a	d	d	a	d	a	d	$witness[4] = 4$
d	a	d	d	a	d	a	d				
			4								shift $= 4$

To capture the duel idea and its implications, consider the scenario below using the same pattern (`daddadad`) as above (the text is `daddadaddadaddadaddadad`, Figure 8 is illustrative as a reference):

					d	a	d	d	a	d	a	d											**pattern**
		d	a	d	d	a	d	a	d														**pattern**
d	a	d	d	a	d	a	d	d	a	d	a	d	d	a	d	a	d	d	a	d	a	d	**text**
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	**position**

Assume j_1 and j_2 are two possible positions in the text for a match. The position of the pattern on the top is $j_1 = 6$, the position of the pattern closest to the text is $j_2 = 3$. Therefore, w in the Figure is 4, because $witness[4]$ is 4 (top pattern is at position 4 with respect to the bottom pattern and $witness[4]$ is 4.) The duel between j_1 and j_2 then proceeds by comparing the $j_1 - 1 + w$ position of the text with the corresponding positions in $pattern$; the observation is that the text position can either match at the w position of the top pattern, or match at the $j_1 - j_2 + w$ position of the bottom pattern or neither (but it cannot match both positions simultaneously). The position that matches wins and the other is discarded. If both lose, then both are discarded. In our example, the text character at positions $j_1 - 1 + w = 9$ is compared with the character at position $w = 4$ of the top pattern and with the character at position $j_1 - j_2 + w = 7$ of the bottom pattern. The bottom pattern is discarded.

Duels are carried out in Step II. The computational structure of Step II is shown in Figure 9. The duels essentially determine a representative candidate for progressively larger blocks of text. Initially the block is small and consists of two characters (2^1). The size of the block representing the winner candidate increases from 2^1 to $2^{\log m - 1}$, and the number of "survivors" decreases by half for each iteration or round. Figure 9 shows the tree structure of the computation for $text = daddadaddadaddadaddadad$ and $pattern = daddadad$. There are $\log m - 1$ levels in the tree, which corresponds to the number of iterations in the algorithm. At each node, the number indicates the position of the winning candidate. The computations are carried out in parallel. After the duels are complete, the remaining candidates are examined in

Figure 9: The binary tree structure of the computation of duels, nodes above level 0 is labeled with position of the winner from a previous round.

Step III. There are at most $2n/m$ remaining candidates. A character by character check is employed to determine which of the candidate positions in the text exactly match $pattern$. Number of comparisons for

this step is $m(2n/m)$, i.e. $2n$, in the worst case and $2n/m$ best case. Pseudo code for steps II and III appear in Appendix A.

The storage requirement for the Vishkin algorithm is $4n + 4m + 7$. The operation count and complexity are described for each step below:

Step I: Preprocessing of the pattern requires $O(m)$ operations and $O(\log m)$ time.

Step II: Duels require $(n/2 + n/4 + \ldots + n/2^{\lfloor \log m \rfloor - 1}) = n(1 - \frac{2}{m}) = O(n)$ operations and $O(\log m)$ time with $O(n/\log m)$ processors.

Step III: Final step requires in the worst case $(mn/2^{\lfloor \log m \rfloor - 1})$ = $2n$ = $O(n)$ operations and $O(\log m)$ time with $O(n/\log m)$ processors. Number of operations in the best case is $(n/2^{\lfloor \log m \rfloor - 1})$ = $2n/m$ = $O(n/m)$.

On a PRAM machine, the algorithm runs in $O(\log m)$ when $\frac{n}{\log m}$ processors are available, otherwise worst case is $3n/p$ and best case is n/p where p is the number of processors. The algorithm does not plan memory access patterns for processors. Further, since the PRAM model assumes that all processors operate in synchronous mode, on actual hardware it is left to the programmer to ensure that processors wait between iterations.

4.2 Sequential Pattern Matching Algorithms

The parallel implementations of the serial algorithms have two steps:

Input: Two one-dimensional arrays: T [1..n] – the text, P [1..m] – the pattern.

Output: All occurrences of the pattern in the text.

Step I: Pre-process the pattern sequentially.

Step II: Split the text into k sections, one section for each processor, then apply the sequential algorithm to each.

4.2.1 The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm is the first linear time sequential pattern matching algorithm. Linearity is achieved by taking advantage of remembering text positions that have been examined from a previous shift of the pattern, thereby avoiding re-examining already known characters. For example, consider matching

the pattern = `ococoe` against the text = `the rococo style`, assuming the current shift is 5 (see Figure 10). Five characters `ococo` in the text have been matched with the characters of the pattern. The next

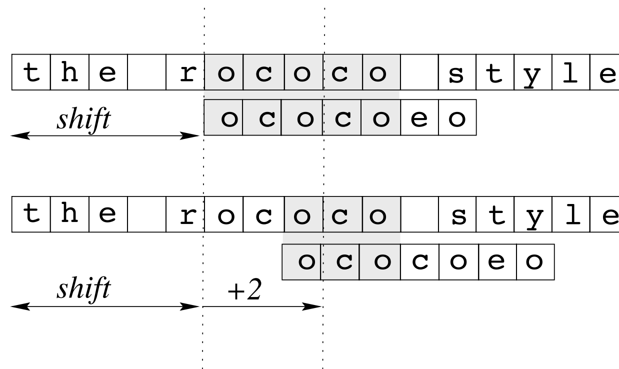

Figure 10: Matching the pattern `ococoe` against a text.

character however, a space, mismatches the pattern character `e`. The Knuth-Morris-Pratt algorithm exploits knowledge of matched characters to determine a new valid shift. In the example, an advancement of one is disregarded because the first character in the pattern `o` would mismatch the already examined character `c` in the text. In contrast, a shift of two utilizes the knowledge of aligning three characters from the text with three characters in the pattern.

The algorithm avoids re-examination of text characters by pre-processing the pattern. A function that determines new shifts is computed by comparing the pattern against itself. The idea is to find the length of the longest prefix of the pattern that is a suffix of characters that have been matched in the text. The difference between the “length” of the suffix already examined and the “length” of the prefix determines the new shift of the pattern. In the example, the difference is 2 because the length of the suffix `ococo` is 5 and the length of the prefix `oco` is 3. A lookup table stores the length of each prefix for each possible suffix. The table for `ococoeo` is shown in Figure 11.

1 2 3 4 5 6 7	suffix length (index of table)
o c o c o e o	pattern
0 0 1 2 3 0 1	prefix length

Figure 11: The prefix lookup table for the pattern `ococoeo`, here a suffix of length 4 has a prefix of length 2.

The algorithm has a worst case running time of $2n + 2m$, where the search phase runs at worst $2n$ and the preprocessing phase runs at worst $2m$. In the best case the number of comparisons in the search

phase is $n + O(1)$ [3]. The worst case running time is independent of the size of the alphabet. The algorithm achieves linearity by remembering text characters that match the characters in the pattern. The Boyer-Moore algorithm described below also takes advantage of tracking mismatched characters.

4.2.2 The Boyer-Moore Algorithm

On average the Boyer-Moore approach is the fastest sequential exact pattern matching algorithm, both in theory and in practice. The key to the algorithm's speed is that it can often skip the examination of text positions. This is accomplished by shifting the pattern from *left* to *right* and examining the pattern against the text *right* to *left* (i.e., the matching process starts by comparing the *last* character in the pattern, as opposed to starting from the beginning of the pattern). Two heuristics help determine how far to the right to shift the pattern on each iteration: (1) the bad-character heuristic and (2) the good-suffix heuristic [7]. The bad-character heuristic uses information from the discovery of bad text characters or uses mismatch positions with the pattern in order to propose a new shift, while the good suffix heuristic proposes a shift that does not cause characters in the current "good suffix" to be mismatched against the proposed new alignment of the pattern. The pattern is then shifted by the largest of the proposed amounts.

To illustrate the effects of the two heuristics, consider the three snapshots of the process in Figure 12. The algorithm is comparing the pattern = `paralegal` against the text = `and his regal seal`. The

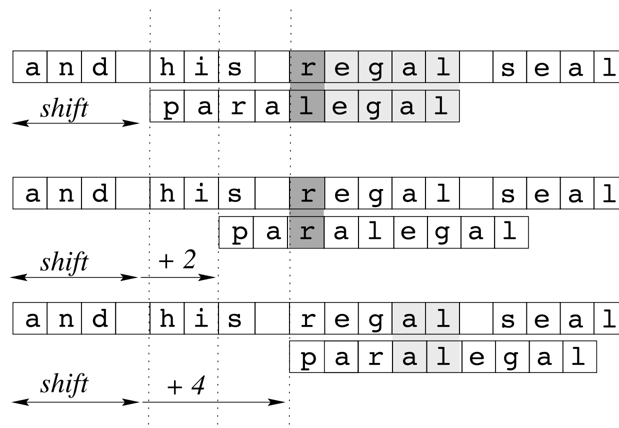

Figure 12: Matching the pattern `paralegal` against a text by comparing characters in a right-to-left manner.

first snapshot shows that the current shift $shift$ is invalid; because the "bad character" `r` does not match the character `l` in the pattern. The second snapshot shows the proposal of the "bad character" heuristic: shift the pattern 2 positions to the right. The "bad character" heuristic proposes this amount so that the discovered "bad character" in the text matches the rightmost character `r` in the pattern (pattern position 3).

The third snapshot shows the proposal of the "good-suffix" heuristics: shift the pattern 4 positions to the right. This heuristic takes advantage of the knowledge of the text characters that matched the rightmost characters of the pattern. It then proposes the smallest right shift of the pattern such that the text characters that previously matched the suffix of the pattern do not mismatch any new characters of the pattern after the proposed shift. Here the heuristic proposes that the pattern shift 4 positions to the right, so that the characters `al` in the text match the `al` characters in the pattern (starting at position 4 of the pattern.) The algorithm then shifts the pattern by 4 since this is the larger of the two proposals.

The worst-case running time, which includes periodic patterns, of Boyer-Moore is $O(n + rm)$, where r is the total number of matches [22]. The worst case thus degenerates to $O(nm)$ comparisons when there are $n - m + 1$ matches. The best case is n/m comparisons and the average is $O(n/m)$ comparisons. Cole shows that if the pattern is non-semi-cyclic[4] the Boyer-Moore algorithm has an upper bound of $3n + \frac{n}{m}$ comparisons [6]. A variant of the Boyer-Moore algorithm, the Galil [14] algorithm, improves the worst case performance to $O(n + m)$. The optimum bound of the expected time complexity of string matching is $O(\frac{\log_q m}{m} n)$, where q is the size of the alphabet [22].

4.3 Discussion of Parallel Pattern Matching Algorithms

The main difference between the serial and Vishkin algorithms is the structure of computation. The Vishkin algorithm uses a binary tree (see Figure 9) while Boyer-Moore and Knuth-Morris-Pratt use a linear list. For the Vishkin computations, the bottom of the binary-tree structure (the leaf nodes) comprises the search text characters, while the internal nodes represent the winner processors "duels". The binary tree can be envisioned as a tournament tree, where the winner represents a potential match and the looser a mismatch. This binary-tree structure reduces computations but as discussed in more detail below, it also increases processor dependencies. A completion of an iteration in the Vishkin algorithm is the same as computing all the winners of duels at a specific level of the tree. At each level or iteration of the tree, the number of processors used decreases by half. As a consequence, an important consideration for the Vishkin algorithm is the extent to which processors must share data (especially when there are fewer processors than in the best case). In the best case of Vishkin's algorithm (when an infinite number of processors are available), the structure implies that at each iteration every processor is dependent on the result of a different processor than in the previous iteration. Initially, this assigns a processor to two character positions in the text at the

[4] A semi-cyclic pattern is of the form wv^k, where w is a proper suffix of v and $k >= 2$.

Algorithm	PRAM Case	Processors	Best Case	Worst Case	Storage Requirement
Knuth-Morris-Pratt	$O(m)$	$n-m+1$	$\frac{n+2m}{p}$	$\frac{2n+2m}{p}$	$n+m$
Boyer-Moore	$O(m)$	$n-m+1$	$\frac{n}{mp}$	$\frac{nm}{p}$	$n+m+c$
Vishkin	$O(\log m)$	$\frac{n}{\log m}$	$\frac{n}{p}$	$\frac{3n}{p}$	$4n+4m+7$

Table 1: Summary of Complexity and Storage Requirement of the Boyer-Moore algorithm and the Vishkin algorithm. The serial algorithms parallel predicted complexity assume linear speed-up. Note, that Vishkin's algorithm runs faster than the serial algorithms even when they are parallelized such that each possible match is assigned a unique processor.

lowest level of the tree.

Processors may also depend on previous processors' results when there are fewer than p processors available, since the PRAM model does not provide guidance on how to assign processors to blocks of text. For example, the text-positions at the lowest level may be assigned round-robin. For instance, if there are four processors, processor 0 is initially assigned text positions 0, 1; processor 1 is assigned positions 2 and 3; processor 3 is assigned positions 4 and 5; processor 1 positions 6 and 7; and so on. This assignment maximizes dependencies among processors, but interestingly the PRAM model does not add a performance penalty for this assignment. On actual parallel machines an assignment of text positions to processors in contiguous blocks is clearly more advantageous with respect to performance.

The algorithms also differ in storage requirements: for the Vishkin algorithm, it is $4n + 4m + 7$; while the Boyer-Moore and Knuth-Morris-Pratt algorithms require $n + m + c$ and $n + m$ amount of storage respectively, where c is the size of the alphabet. The storage for the Vishkin algorithm is $2n$ for a matrix that holds the tree structure, a vector of size n that holds boolean values indicating if there is a potential match or not, and n for the text itself. The Boyer-Moore algorithm's storage requirements include n for the text, c for the bad-character heuristics, and m for the good-suffix heuristics. The Knuth-Morris-Pratt algorithm uses n for storing the text and m to store the prefix table. Table 1 is a summary of the complexity and the storage requirements of the serial and Vishkin algorithms.

5 Parallel Implementations

All three algorithms were implemented using the $Cthreads$ [27] library for parallel user-level threads. The C-threads library is ported across different platforms (this research uses the KSR-2, the SGI Power Challenge and the SGI Origin 2000), which results in the same user code executed across the various platforms used in our experimentation. Although C-threads allows several threads to be mapped to the same processor, that feature is not needed and not used, as threads are mapped to unique processors throughout the experiments examined in this paper.

5.1 Parallel Implementation of the Serial Algorithms

Parallel versions of the Boyer-Moore and the Knuth-Morris-Pratt algorithms are created by converting the sequential routine into a function that can operate on sub-parts of the text. Simple domain decomposition is used such that the search text is divided into k sub-parts when k processors are available. Each thread then proceeds to process its sub-part of the text. In the Boyer-Moore algorithm, each thread keeps its own

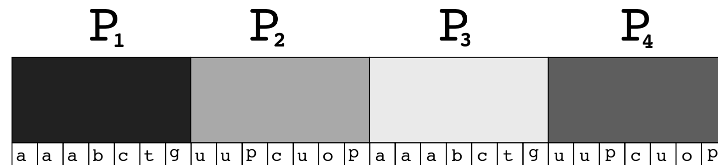

Figure 13: Processors are assigned are assigned continuous subparts of the text and work independently in the parallel implementations of the serial algorithms.

local copy of the two specialized data structures used to implement the two heuristics. The size of the array required for implementing the good-suffix heuristic is the same size as the pattern, namely m. The data structure to implement the bad character heuristics is the size of the ASCII character set. Each thread in the Knuth-Morris-Pratt algorithm only keeps its own copy of the prefix lookup table, which is of size m. The vector containing the search text is shared between threads both in the Knuth-Morris-Pratt and Boyer-Moore algorithms. Pseudo code for the text analysis part of the algorithms appears in Appendix B.

5.2 Implementation of Vishkin's Algorithm

The algorithm given by Vishkin is written under the assumption that there are $n/\log m$ processors. Few characters are assigned per processor when the pattern is small. For example, when the pattern is of length 4, then each processor is assigned two characters of text. The KSR-2, however, is limited to 64 processors, so even a modest text requires some type of decomposition to distribute the problem between processor elements.

In our first implementation of the Vishkin algorithm, computations are assigned to processors round-robin. Here, processing of text is interleaved between the processors, so that each thread determines the winner of the partition numbered by its ID. Then the thread proceeds to determine the winner of the partition $ID + num_threads$, and so on. When a thread completes a level and is ready to advance to the next level, it must pause until other threads complete their processing for the current level. This is because computations at one level depend upon other threads' results at the next lower level. Conversely, the PRAM model assumes synchronous machine operation which implies that all threads naturally complete each stage

at the same time. The structure of the computation scheme is shown in Figure 14. The figure shows a subtree of the computational tree. Different shades of gray denote assignments to different processors.

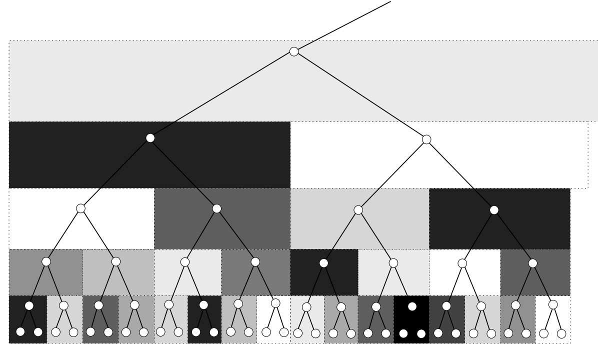

Figure 14: A subtree of the computational tree, with different shades of gray denoting assignments to different processors.

The main data structures given by the algorithm are: *left*, a multi- dimensional array, a vector *match*, and a vector *text*. The *left* matrix holds the position of the winning candidates of "duels", as discussed in Section 4.1. The *match* vector is indexed by the character position corresponding to the search text and contains a boolean value which indicates if the specified position can still be considered as a possible candidate or not. The *text* array contains the search text. Threads share data in $match$, $left$ and *text*; each thread has its own copy of the vector $pattern$ and the vector $witness$.

Pseudo code for the implementation is given in the appendix in three modules (see Figure 26 in Appendix A). The first module called "thread" synchronizes between levels, the second module called "work on level" is what each thread executes at each level, the third called "check candidates" is the final step, where each surviving candidate is checked in parallel. The resulting computational tree is shown in Figure 14.

6 Performance Results

The first set of experiments were run on a 64 processor KSR2 supercomputer. In each plot a data point represents the average of over 50 runs. Figure 15 shows how performance varies as 1 to 40 processors search a 10 megabyte text and a pattern size of 10 using the Boyer-Moore and Vishkin algorithms. For these performance measurements, timing starts after pre-processing is completed and stops after the last processor finishes processing. For the Vishkin algorithm the performance initially improves dramatically (up to 5 processors), then levels off and degrades. This is due to shared memory contention. For n processors, in this implementation, each segment of n characters in the text is accessed by all n processor. Since n is never

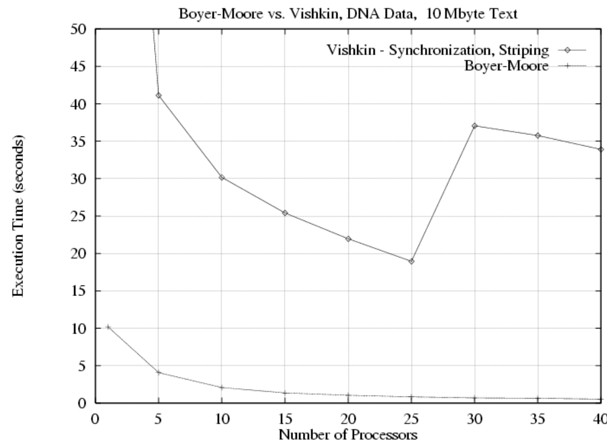

Figure 15: Comparison between the algorithms of Boyer-Moore and Vishkin on the 64 processor KSR2 Supercomputer. Lower values indicate better performance.

greater than 40, every sub-page (128 bytes on the KSR-2) of the text must migrate to each processor. This suggests that Vishkin's algorithm will usually run at the maximum memory/bus speeds of this machine particularly since the pattern matching operation are not very computationally expensive. An interesting aspect of performance of Vishkin's algorithm, depicted in Figure 15, is the significant degradation for 25 and more processors, where at least one processor has migrated to another ring (the KSR processors are arranged in two rings of 32 processors each). The cost of inter-ring memory access is much higher than intra-ring access, thereby leading to a significant reduction in the algorithm's performance.

The Boyer-Moore algorithm performs significantly better than the Vishkin algorithm, which suggests that for a super computer with 64 processors it may be more beneficial to use a sequential algorithm parallelized by domain-decomposition rather than an optimal PRAM algorithm. Furthermore, if memory is limited, then the Boyer-Moore algorithm is again preferable because the Vishkin algorithm requires significantly more memory than the Boyer-Moore algorithm since it needs additional data structures that have the same relative sizes as the length of the text.

In summary, these results indicate that inter-processor dependencies can significantly impact performance on real machines and invalidate PRAM predictions (PRAM predicts that Vishkin should perform better than Boyer-Moore). In particular, memory access patterns must be planned carefully, especially when text is forced to migrate between rings as in the KSR implementation of the Vishkin algorithm. Simple domain decomposition that reduces remote accesses can be beneficial as the Boyer-Moore algorithm demonstrates.

7 Improving the Performance of Vishkin's Algorithm

In the first implementation, performance suffers because fine-grained interleaving of input data requires each processor to cache all portions of the text. We reduce remote memory accesses and paging by decomposing the data into continuous partitions and dividing them among the processors. In this revised

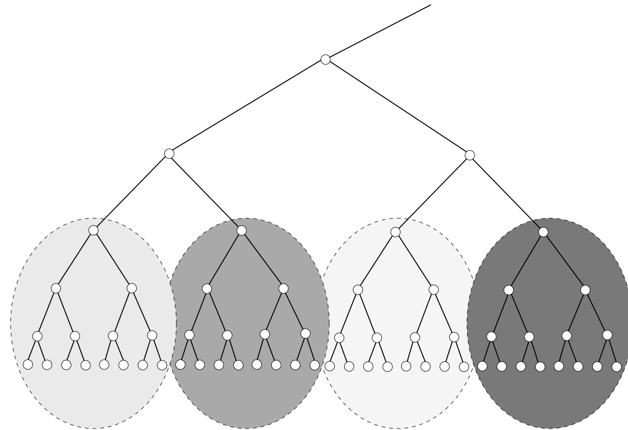

Figure 16: The computational tree, in the revised Vishkin approach.

implementation of Vishkin's algorithm, each thread is assigned disjoint sub-trees of the computation tree, as illustrated in Figure 16. Here, a different shade of gray indicates an assignment of computation to a different processor. This scheme offers two improvements over the initial implementation:

- Threads are *not* dependent upon results of other threads.
- Processors access fewer memory pages because they work on smaller, contiguous portions of the text.

To save memory, the size of the $left$ matrix is reduced to contain only two levels of the computation tree. If the algorithm requires more than two levels, then the previous result from an earlier round is overwritten (measurements without optimizing memory usage have slightly worse performance). Pseudo code is listed in Appendix A. Performance results appear in Figure 17.

A substantial improvement is realized by manipulating the memory reference scheme; performance is better in all cases for this implementation versus the interleaved approach. There is also no performance reduction as the number of processors exceeds 30 on the KSR machine as occurred previously. This is because inter-ring memory access is not usually necessary when processors work individually on their own sub-section of text, although there are still pages shared among processors for the *left* array. Namely, if a thread does not need access to the exact same position recently written by another processor, it may need

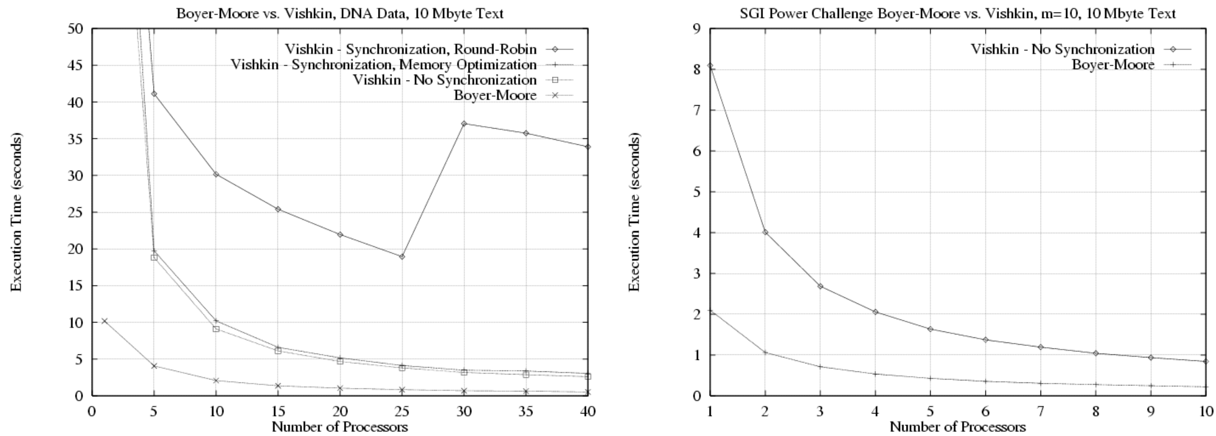

Figure 17: The plot on left demonstrates the performance of Vishkin with domain decomposition and memory optimization on the KSR2 supercomputer. The plot on the right demonstrates improved performance of Vishkin with domain decomposition on the SGI Power Challenge. The performance of the Vishkin algorithms are compared against the performance of the Boyer-Moore algorithm on both machines. Lower values indicate better performance.

access to a nearby position.

Even better performance may be realized for the Vishkin algorithm by eliminating synchronization, at the cost of additional memory for the $left$ matrix. By extending its size, we remove the possible conflict of processors accessing the same memory cell, and thus the requirement for synchronization. The performance results for this scheme are also shown in Figure 17. Again, performance is improved in all cases compared to the previous implementations, but the gains are not as significant, because even though synchronization cost is decreased there are additional memory management costs (such as paging costs).

The Boyer-Moore algorithm is also impacted by memory access and synchronization factors. Speedup [5] of the Boyer-Moore implementation for 10-40 processors on the KSR2 is shown in Figure 18: each search for a 10 byte pattern in texts varies from 1 to 10 megabytes. For small sizes of text the costs of cthread_fork(), barrier(), and the allocation of parameter space for each of the processors dominate the overall computational cost, but as the size of the text increases this effect becomes less significant. Although each processor operates on independent sections of the text, they must all contend for access to the bus since the text is in shared memory. This contention results in less than linear speedup for large numbers of processors and small texts. As the text becomes large, however, speedup approaches linearity for larger numbers of processors.

Finally, Figure 19 shows the comparison of run times for the Vishkin and Boyer-Moore implementations as the size of the text is varied. Although both show approximately linear performance, it is safe to say

[5] the ratio between the sequential execution time (of a pure sequential implementation of the same algorithm), and the the execution time of the parallel implementation

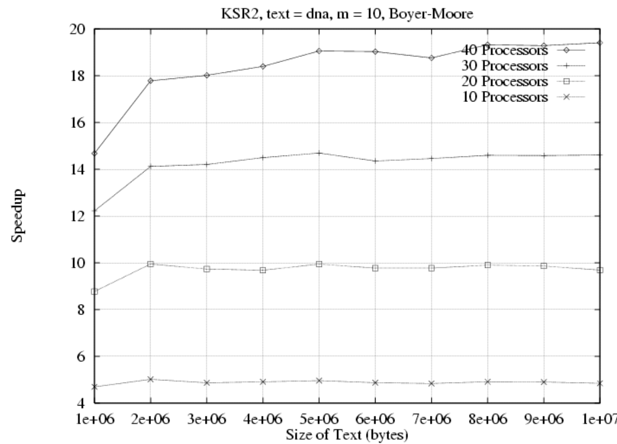

Figure 18: Speedup of the Boyer-Moore implementation on the KSR2. Higher values indicate better performance.

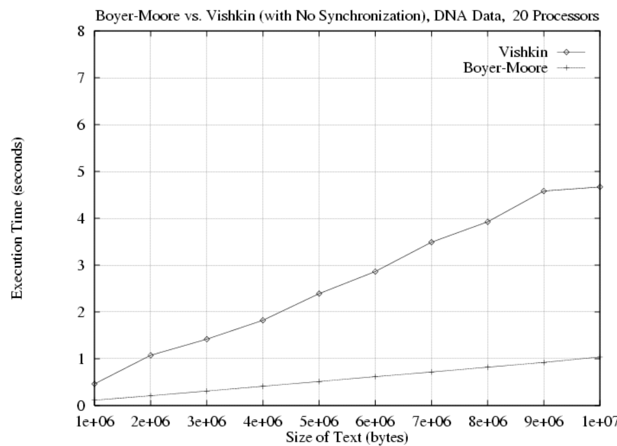

Figure 19: Comparison of the Boyer-Moore and the improved Vishkin algorithm on the KSR2. Lower values indicate better performance.

that the Boyer-Moore implementation is significantly faster than the Vishkin implementation for all sizes of text.

The performance results suggest that communication cost between processors, both for synchronization and memory accesses, is the main bottleneck for performance. It is doubtful that algorithms like Vishkin's, which exploit communication between processors instead of mapping contiguous blocks of data to one processor, can ever out-perform a sequential algorithm parallelized by techniques like domain decomposition on contiguous data. Furthermore, as CPU speeds and memory speeds diverge, locality considerations become even more important. This suggests the increasing importance of augmenting theoretical parallel models like PRAM to take into account memory effects. Such memory effects are explored in more detail in the next section.

8 Varying the Size of Data Assignment for Interleaving

When the text is large, the Vishkin algorithm requires more processors than are available on existing hardware. For example, searching for an eight byte pattern in a 10 MB byte text requires more than 3 million processors. Even if there were enough processors, performance may be affected by memory access and synchronization costs.

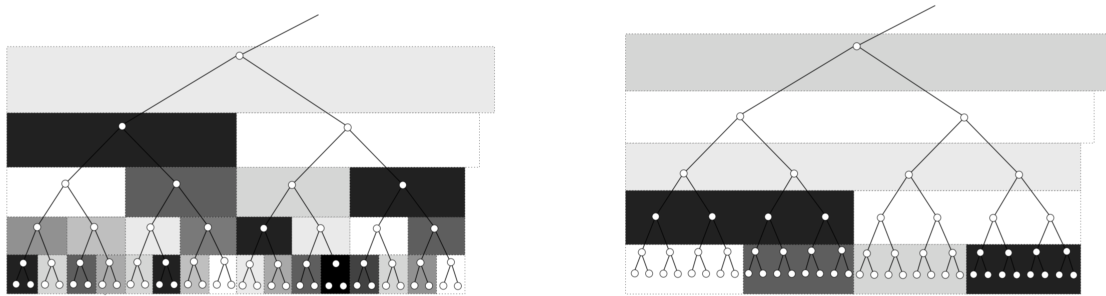

Figure 20: Changing the interleave factor for the Vishkin implementation, an interleave size of 1 is shown on the left and an interleave size of 4 is shown on the right.

Access patterns of a larger number of processors than exist in actual hardware can be simulated by manipulating the assignment of data to real processors. Here, a physical processor represents multiple processors required by the Vishkin algorithm. The general idea is to simulate the number of characters that are assigned to each processor in the initial stage of computation. For example, when the text is 1 MB, or 1,048,576 bytes, and the pattern is 8 bytes, processors compute fragments of text that are between 2-3 characters in length. In terms of Vishkin's computational tree, this corresponds to one computational node assigned to one processor, while the neighboring (sibling) nodes on the same level in the tree are assigned to a different physical processor.

To simulate this behavior, we study the impact of performance when different sizes of text fragments are assigned to processors round-robin (i.e. a processor is "re-used" in round robin fashion when the algorithm calls for more processors than are available). A subtree of the computational node assignment to processors for searching for an 8 byte pattern in text of 1 MB is shown on the left in Figure 20. Recall that for patterns of size 8 the Vishkin algorithm computes the first 3 levels in the computational tree (leaf and two interior nodes), and that the computation does not continue to the root of the tree. The node assignment for a pattern of size 128 is shown on the right of Figure 20. Shades of gray indicate node assignments to different processors. A round-robin assignment ensures that text-fragments assigned to nearest neighbors in sibling nodes are on different processors. This emulation of a larger system is optimistic in the sense that the real

access costs higher if the memory hierarchy is larger. For example, in the KSR architecture we experienced a memory penalty when processors accessed data residing on different rings.

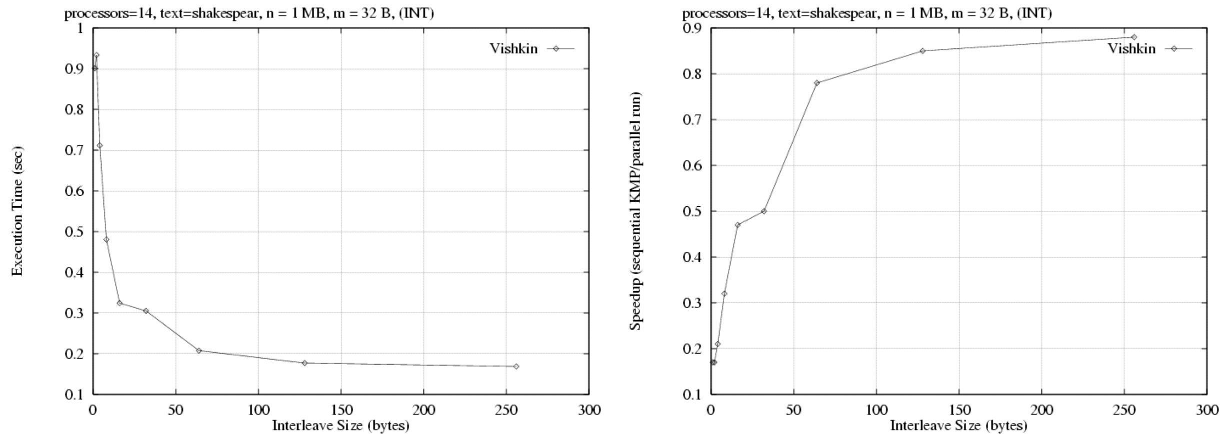

Figure 21: Changing Interleaving: Execution time on the left (lower values indicate better performance) and speedup on the right (higher values indicate better performance). Performance on the SGI Origin 2000.

To implement the access pattern of Vishkin in a large system, we re-examine our initial implementation that assigned the computational nodes round-robin. The scheme is modified so that each thread can work on a specified interleave size. When the interleave size is 4 and there are 2 processors available, a thread first works on 4 neighboring nodes then it skips 4 nodes to work on its next set of nodes to simulate a different processor (an interleave size of four is shown on left in Figure 20).

Performance using 14 processors on an SGI Origin 2000 searching for a pattern 32 bytes long in English text of size 1 MB is shown on the left in Figure 21. Each point represents the median of 40 runs. Additional experiments using different sizes of text, multiple patterns and alphabets yielded similar results. The interleave factor is varied from 254 to 4. A factor of 4 simulates systems with an access-pattern of 2 nodes per processor, the actual allotment in a CREW-PRAM for a pattern of 32 bytes. As the interleave factor decreases, performance degrades exponentially. The ratio of the sequential Knuth-Morris-Pratt algorithm with the Vishkin algorithm while varying the interleave factor of the Vishkin algorithm is shown on the right in Figure 21. The text is 1 MB, the pattern is 32 bytes and 14 processors are used for the parallel run. The results show a severe degradation of speedup as the interleave decreases. For these interleave factors the speedup is never greater than 1.

The memory-centric results presented in this section suggest that performance will *degrade* instead of improve as processors are added to the system. This is because locality of data becomes poorer as the number processors is increased: a processor depends on a larger number of other processors to compute its

results. In addition, since the assignment of data fragments to processors becomes smaller, false sharing increases and further worsens performance. Additional synchronization costs not demonstrated in these results will impact performance further, since in our simulation only physical processors synchronize between levels in the computational tree. As an interesting side-note, we observed that the performance of Vishkin with an interleave factor 254 performed similarly to the Vishkin implementation without any synchronization primitives. This is probably because the impacts of false sharing and locality of data become less significant when the interleave factor is large. Each processor can work more independently as the interleave factor increases.

The type and size of the pattern and text to be searched also impact performance. To verify that changes in these factors are not so significant as to reverse the importance trends noted so far, additional experiments were run by varying these parameters. The next two subsections include the results from these experiments.

8.1 Different Types of Text

The size of the alphabet also impacts performance of the Vishkin algorithm. Performance results with two different alphabet sizes are shown in Figure 22. The alphabet size is 85 on the left and 2 on the right. The text with the larger alphabet is several copies of the complete works of William Shakespeare (provided by Project Gutenberg Etext of Illinois Benedictine College). The text consists of symbols that can be found in a file containing English text, i.e. spaces, return characters, letters, punctuation etc. The pattern is 16 bytes and there are 0.0001% Matches of the pattern in the text (100% Matches implies there is a match at every position of the text). The text with the smaller alphabet contains 2 letters: a's and b's. The text contains 11.6% Matches of the pattern.

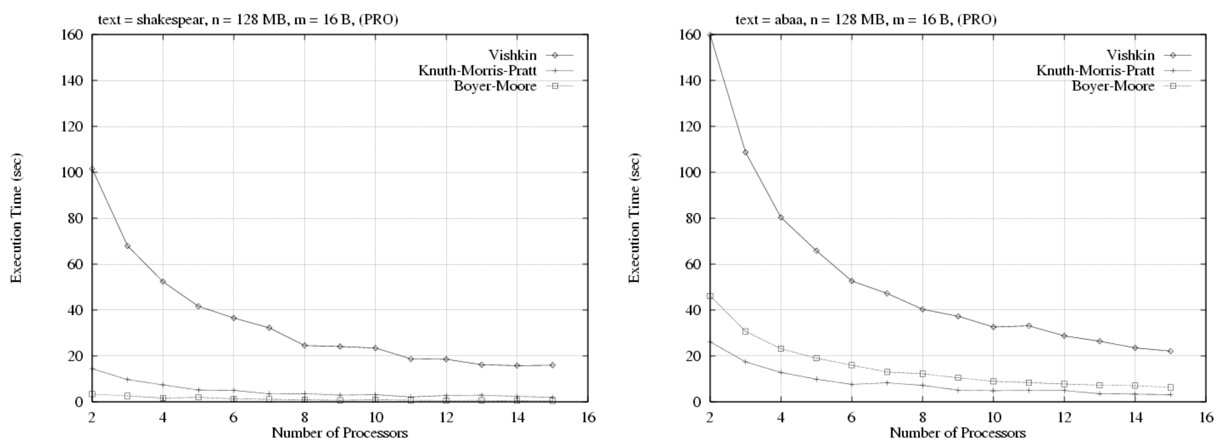

Figure 22: Different type of text: English text with 0.0001% matches on the left and periodic text with 11.6% matches on the right. performance on the SGI Origin 2000. Lower values indicate better performance.

Vishkin performs better with a larger alphabet size. This is because when using a larger alphabet it is more likely to eliminate possible candidates. In other words the probability of a match is lower when the alphabet is larger. The Boyer-Moore and the Knuth-Morris-Pratt algorithms also perform better when there is a lower probability of a match. Note that Knuth-Morris-Pratt out-performs Boyer-Moore when the alphabet size is small. In experiments on random text (with fewer matches than English text), the performance of all three algorithms is similar but slightly better than their performance for English text because they are more likely to eliminate candidates. The performance of random text is similar in performance to the previous plots and in shown in Figure fig:text-perf-rand.

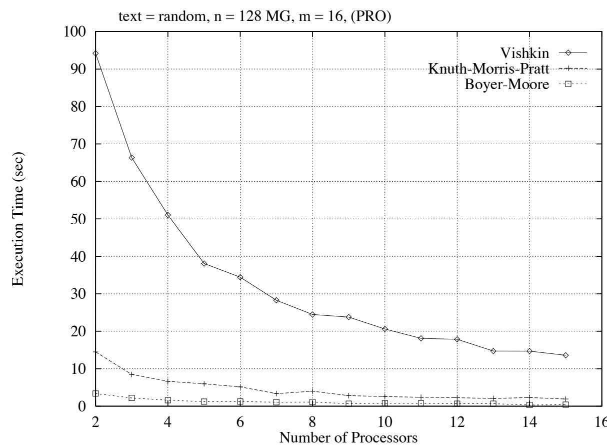

Figure 23: Random text. Performance on the SGI Origin 2000. Lower values indicate better performance.

8.2 Pattern Sizes

The size of the pattern significantly impacts run time. When the pattern is large in relation to the text, performance is significantly degraded in the parallel Boyer-Moore and the Knuth-Morris-Pratt implementations (see Figure 24). There is at least one factor responsible for the reduced performance. In the sequential version of the Boyer-Moore and the Knuth-Morris-Pratt algorithms, an earlier shift determines the length of the next shift. But the parallel implementation loses this information for different segments on different processors. This loss may be significant for large patterns For large patterns, the performance of the Vishkin implementation is somewhat close to the performance of the parallelized sequential algorithms. This is because a candidate match is more likely to lose a duel with larger patterns. For small patterns (3 bytes and smaller) the Vishkin algorithm performs poorly because no duels are performed and every position in the text is treated as a possible match. The Boyer-Moore algorithm performs better when the pattern is larger than 4 bytes.

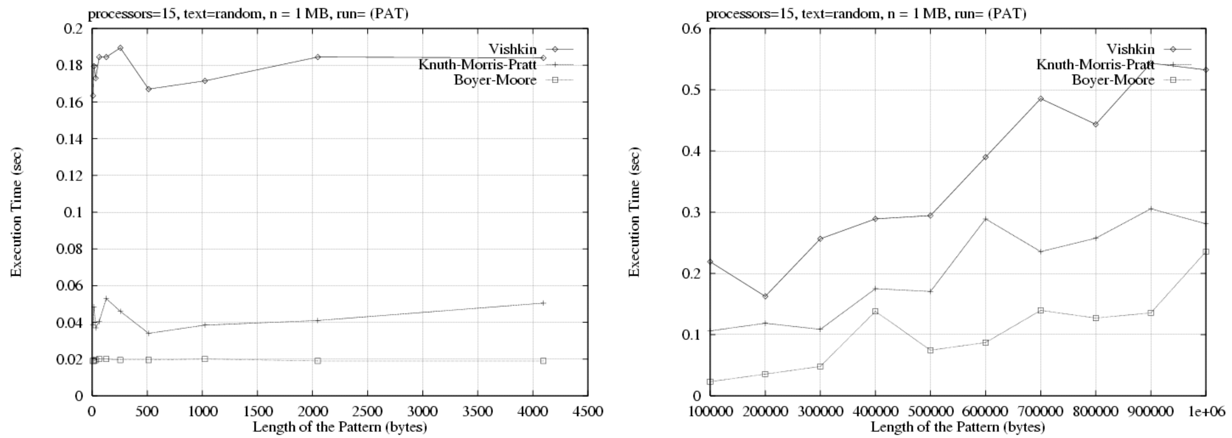

Figure 24: Changing pattern sizes: small patterns on left and large patterns on right. Performance on the SGI Origin 2000. Lower values indicate better performance.

9 Discussion and Conclusions

The most significant result of this research is that use of the PRAM model for performance prediction leads to incorrect statements about the suitability of various parallel algorithms for existing and future parallel architectures. This is particularly the case for fine-grained parallel algorithms where the PRAM model's predictions do not currently hold (and are not likely to hold in the future), as demonstrated by the performance measurements presented in this paper. These measurements indicate the need for more realistic models that account for delays in memory access, synchronization and communication costs.

Additional results of this work compare parallelized versions of the Boyer-Moore and Knuth-Morris-Pratt sequential algorithms with Vishkin's parallel algorithm. We study algorithms performance on several different parallel architectures using DNA sequences, random and English text. Performance of both the Boyer-Moore and the Knuth-Morris-Pratt implementations are shown to be significantly better than that of the Vishkin implementation.

Detailed measurements indicate that inter-processor dependencies significantly impact algorithmic performance on actual machines. This invalidates PRAM based predictions of algorithmic performance, as PRAM determines that Vishkin is optimal while Boyer-Moore parallelized by domain decomposition is not. In particular, our results show that memory access patterns must be planned carefully, especially when text is forced to migrate between rings as in the KSR implementation of the Vishkin algorithm. Appropriate domain decomposition that reduces remote accesses can be beneficial as demonstrated by alternative implementations of Vishkin's algorithm and by parallel implementations of the Boyer-Moore and the Knuth-Morris-Pratt algorithms. These results hold for studies of parallel implementations of Vishkin's,

Boyer-Moore's and Knuth-Morris-Pratt's algorithms in which the impact of several factors on their relative performance is investigated including:

- type of text (English, Random, DNA),
- size of text,
- size of pattern and
- alternative assignments of segments of text to processors.

In all cases, the sequential algorithms (by Boyer-Moore and Knuth-Morris-Pratt) parallelized by domain decomposition outperform Vishkin's theoretically optimal (as predicted by PRAM) algorithm. This result remains true even when the Vishkin algorithm's implementation is optimized to account for these factors.

Furthermore, our performance studies of Vishkin's PRAM optimal algorithm suggest that performance will degrade instead of improve as processors are added to the system. This is because data is less likely to be local to the processor that needs it as the number of processors increases. As locality worsens, sequential bottlenecks increase and false sharing between processors increases as well.

We conclude that the PRAM model does not accurately predict algorithmic performance, due to its assumption of uniform memory access for an unlimited number of processors with no penalties for memory contention. This assumption is in direct conflict with current directions in hardware design where in fact, CPU speed improvements continue to outpace gains in memory speeds. Furthermore, manufacturers are building shared memory machines with uniform memory access with fewer processors (e.g. the SGI Power Challenge), while hardware and software efforts are directed at machines with more processors utilizing distributed memory schemes, again implying the impracticality of "common" memory with a large number of CPUs. Neither case – few processors and fast shared memory, or many processors with slow shared memory – matches the implicit assumptions of the PRAM.

Acknowledgment

We are grateful to George Riley who ported the Vishkin and Boyer-Moore implementations to the SGI Power Challenge.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, CA, October 1987. IEEE Computer Society Press.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[3] R. A. Baeza-Yates. String searching algorithms. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structure and Algorithms*, pages 219–240. Prentice-Hall, 1992.

[4] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26, August 1992.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[6] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM Journal on Computing*, 23(5):1075–1091, October 1994.

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

[8] Maxime Crochemore, Leszek Gąsieniec, Ramesh Hariharan, S. Muthukrishnan, and Wojciech Rytter. A constant time optimal parallel algorithm for two-dimensional pattern matching. *SIAM Journal on Computing*, 27(3):668–681, June 1998.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E.Santos, R. Subramonian, and T. van Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[10] D. Culler, R. M. Karp, D. Patterson, A. Sahay, E. Santos, K. E. Schauser, R. Subramonian, and T. van Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[11] Y. Feldman and E. Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, 35(10):60, October 1992.

[12] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, California, 1–3 May 1978.

[13] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1994.

[14] Z. Galil. On improving the worse case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, September 1979.

[15] Z. Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67(1–3):144–157, October/November/December 1985.

[16] C.N Galley. A CRCW PRAM lower bound for problems on two dimensional arrays. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 27:77–95, June 1998.

[17] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, New York, 1988.

[18] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, Arlington, Virginia, 23–25 January 1994.

[19] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design*. Morgan Kaufmann Publishers, San Francisco, 1997.

[20] J. F. JáJá and K. W. Ryu. The block distributed memory model. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):830–840, August 1996.

[21] Marek Karpinski and Wojciech Rytter. Alphabet-independent optimal parallel search for three-dimensional patterns. *Theoretical Computer Science*, 205(1–2):243–260, 28 September 1998.

[22] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[24] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 241–251, Denver, CO, USA, 1997. ACM Press.

[25] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 3–17, Seattle, WA, October 1996. USENIX.

[26] M-C. Rosu, K. Schwan, and R. M. Fujimoto. Supporting parallel applications on clusters of workstations: the intelligent network interface approach. In *Proceedings of the 1997 IEEE High Performance Distributed Computing*, August 1997.

[27] K. Schwan, H. Forbes, A. Gheith, B. Mukherjee, and Y. Samiotakis. A C thread library for multiprocessors. Technical Report GIT-CC-91-02, Georgia Institute of Technology. College of Computing, 1991.

[28] Inc Silicon Graphics. Power Challenge technical report. Technical report, Silicon Graphics, Inc, 1996.

[29] Inc Silicon Graphics. Performance tuning optimization for Origin2000 and Onyx2. Technical Report 007-3511-001, Silicon Graphics, Inc, 1997.

[30] C. Stanfill. Parallel information retrieval algorithms. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structure and Algorithms*, pages 459–497. Prentice-Hall, 1992.

[31] E. Ukkonen. On approximate string matching. In M. Karpinski, editor, *Proceedings of the International Conference on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 487–495. Springer Verlag, August 1983.

[32] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[33] L. G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8):103–111, August 1990.

[34] U. Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67(1–3):91–113, October/November/December 1985.

[35] U. Vishkin. Can parallel algorithms enhance serial implementation? *Communications of the ACM*, 39(9):88–91, September 1996.

A Vishkin's Algorithm

The array $match$ indicates if there there is a match in the text at a specified position. The $left[i][j]$ array stores the outcome of the duels; i indicates which level is being evaluated, and j indicates the particular block within the given level. The array contents reflect the position of the representative for the block. The implementation of Vishkin's algorithm is influenced by Gibbons and Rytter's [17] description of the algorithm.

```
function DuelText (j1, j2)
    if ( j1 = mismatch ) return j2
    if ( j2 = mismatch ) return j1
    else
        w ← witness[j1 − j2 + 1]
        z ← text[j1 − 1 + w]
        y ← pattern[j1 − j2 + w]
        x ← pattern[w]
        if ( z ≠ y )
            match[j2] ← FALSE
        if ( z ≠ x )
            match[j1] ← FALSE
        if ( match[j2] = TRUE )
            return j2 as surviving candidate
        else if ( match[j1] = TRUE )
            return j1 as surviving candidate
        else
            return NULL
    end else
end DuelText

procedure ElliminateCandidates
    for k ← 1 to log m − 1 do
        for all ( blocks 'a' at level 'k' ) par do
            j2 ← left[k][2a − 1] /* left child of "a" at level k */
            j1 ← left[k][2a]     /* right child of "a" at level k */
            left[k][a] ← DuelText(j1, j2)
        end_for all
    end_for
end_procedure

procedure FindTrueMatch
for all j ← first_block to last_block par do
    candidate = left[ last_level ] [j]
    if ( candidate ≠ NULL )
        for all i ← 1 to m
            check letter by letter
            if ( a mismatch is found )
                match[candidate] = FALSE
                break out of for loop
            else if ( looked at whole string and a match )
                record a MATCH at position candidate(j)
            end_if
        end_for all
    end_if
end_for
```

Figure 25: Procedures for Vishkin's pattern matching algorithm.

```
procedure Thread
for level = 1 to log m − 1 do
        WorkOnLevel()
        increment (numbers_done)
        if ( last to be done on “level” )
                reset number_done to 0  mutual exclusion
                increment (level_working on)
        else
                wait for other processors to finish
end for

procedure WorkOnLevel
a = my_processor_number
while ( this level is not done ) do
        j2 = left[(k-1) mod 2] [2a - 1]  left child of a at level k
        j1 = left[(k-1) mod 2] [2a]  right child of a at level k
        left [k mod 2] [a] = DuelText(j1, j2)
        a = number_of_processors + a
end_while

procedure CheckCandidates
for j = start_block to end_block do
        candidate = left[ last_level ] [j]
        if  ( candidate ≠ NULL )
                while ( still a match for this position )
                        check letter by letter
                        if  ( a mismatch is found )
                                match[candidate] = FALSE
                                break out of while loop
                        else if  ( looked at whole string and a match )
                                found a MATCH
                        end_if
                end_while
        end_if
end_for
```

Figure 26: First approach in the implementation Vishkin algorithm.

```
num_trees = number of subtrees
if ( participate )
        num_blocks = (num_trees/num_procs) * 2^(log m-1)
        num_blocks = num_block/2
        start_block = (my_id - 1) * num_blocks + 1
        if ( MyPE = LAST )
                end_block = num_trees * 2^(log m-2)
        else
                end_block = start_block + (num_blocks - 1)
        for level = 1 to log m - 1 do
                for a = start_block to end_block do
                        j_2 = left[(k-1) mod 2] [2a - 1]
                        j_1 = left[(k-1) mod 2] [2a]
                        left [k mod 2] [a] = duel_text(j_1, j_2)
                end_for
                if ( k ≠ log m - 1)
                        num_blocks = num_blocks/2
                        end_block = end_blocks/2
                        start_block = end_blocks - num_blocks + 1
                end_if
        *       increment numbers_done mutual exclusion
        *       if ( last to be done on "level" )
        *               reset number_done to 0  mutual exclusion
        *               increment (level_working on)
        *       else ( wait for other processors to finish )
        end_for
end_if
```

Figure 27: Second approach in the implementation Vishkin algorithm.

B Boyer-Moore's Algorithm

```
procedure Boyer-Moore
    shift = 0
    last_shift = n - m
    while (shift ≤ last_shift) do
        j = m
        while ( (j > 0) and (pattern[j] = text[shift + j] ) ) do
            j = j - 1
        if ( j = 0 ) then
            found a MATCH
            shift = shift + GoodSuffix[0]
        else if ( GoodSuffix[j] > (j - BadSuffix[ text[shift + j]] ) then
            shift = shift + GoodSuffix[j]
        else
            shift = shift + (j - BadSuffix[ text[shift + j]])
    end_while
end_procedure
```

Figure 28: Procedure for Boyer-Moore's pattern matching algorithm.

C Knuth-Morris-Pratt's Algorithm

```
procedure Knuth-Morris-Pratt
    shift = 1
    last_shift = n
    q = 0
    for i = shift to last_shift do
        while ( (q > 0) and (pattern[q+1] ≠ text[i]) ) do
            q = prefix[q]
        if ( pattern[q+1] == text[i] )
            q = q + 1
        if ( q == m )
            found a MATCH
            q = prefix[q]
    end_while
end_procedure
```

Figure 29: Procedure for Knuth-Morris-Pratt's pattern matching algorithm.