Experiments With List Ranking for Explicit Multi-Threaded (XMT) Instruction Parallelism (extended abstract) *

Shlomit Dascal and Uzi Vishkin

Abstract. Algorithms for the problem of list ranking are empirically studied with respect to the Explicit Multi-Threaded (XMT) platform for instruction-level parallelism (ILP). The main goal of this study is to understand the differences between XMT and more traditional parallel computing implementation platforms/models as they pertain to the well studied list ranking problem. The main two findings are: (i) Good speedups for much smaller inputs are possible. (ii) In part, this finding is based on competitive performance by a new variant of a 1984 algorithm, called the No-Cut algorithm. The paper incorporates analytic (*non-asymptotic*) performance analysis into experimental performance analysis for relatively small inputs. This provides an interesting example where experimental research and theoretical analysis complement one another. [1]

Explicit Multi-Threading (XMT) is a fine-grained computation framework introduced in our SPAA'98 paper. Building on some key ideas of parallel computing, XMT covers the spectrum from algorithms through architecture to implementation; the main implementation related innovation in XMT was through the incorporation of low-overhead hardware and software mechanisms (for more effective fine-grained parallelism). The reader is referred to that paper for detail on these mechanisms. The XMT platform aims at faster single-task completion time by way of ILP.

* Partially supported by NSF grant CCR-9416890 at U. Maryland.

[1] This paper suggests a possible new utility for the developing field of algorithm engineering; contribute to the interplay between two research questions: how to build new computing systems? and how to design algorithms?
More concretely, reducing the completion time of a single general-purpose computing task by way of parallelism has been one of the main challenges for the design of new computing systems over the last few decades. ILP has been a realm where success has already been demonstrated. A motivation for originally proposing XMT has been to provide a platform for the new utility in the context of the ILP realm.
The referees asked us a few general questions: (i) Does it make sense to compare speed-ups for XMT, which for now is nothing but a simulator, with speed-ups for a real machine? Noting that such comparisons are standard in computer architecture research, we agree that caution is required. (ii) Why is the XMT model not more succinctly presented? The XMT model is based on an assembly language and an execution model since this is the way it will be implemented. This is similar to the the underlying assumptions in measuring ILP in Section 4.7 in [HP96]. (iii) How is bandwidth incorporated in the XMT model? Bandwidth does not explicitly appear; within a chip it is generally not a bottleneck, see [DL99]).

1 Introduction

This work considers practical parallel list-ranking algorithms. The model for which programs are written is a single-program multiple-data (SPMD) "bridging model". This model is designated as a programmer's model for a fine-grained computation framework called Explicit Multi-Threading (XMT), which was introduced in [VDBN98]; the XMT framework covers the spectrum from algorithms through architecture to implementation; it is meant to provide a platform for faster single-task completion time by way of instruction-level parallelism (ILP). The performance of XMT programs is evaluated as follow: the performance of a matching optimized XMT assembly code is measured within an XMT execution model. (We use in the current paper the so-called *Spawn-MT* programming model - the easier to implement among the two programming models presented in [VDBN98]). The XMT approach deviates from the standard PRAM approach by incorporating *reduced synchrony* and departing from the lock-step structure in its so-called *asynchronous mode*. Our envisioned platform uses an extension to a standard serial instruction set. This extension efficiently implements PRAM-style algorithms using explicit multi-threaded ILP, which allows considerably more fine-grained parallelism than the previously studied parallel computing implementation platforms/models.

The list ranking problem was the first problem considered as we examined and refined many of the concepts in the XMT framework. The problem arises in parallel algorithms on lists, trees and graphs and is considered a fundamental problem in the theory of parallel algorithms. Experimental results are presented.

Empirical study of parallel list ranking algorithms Implementation of parallel algorithms for list ranking has been considered by many including [HR96], [Re94], [SGS], [Si97a] and [Si97b]; Section 7 gives more detail. The three main observations in this paper are as follows: (i) *Good speedup relative to the corresponding serial algorithm are possible for smaller input sizes than before.* We are not aware of previous results for this problem which are competitive with the counterpart serial algorithm for input size not exceeding 1000; often applicable input sizes have been much larger. By efficiently supporting lighter threads, the XMT envisioned implementation platform made this possible. List ranking is a routine which is employed in many large combinatorial applications; we proceed to illustrate the *importance of achieving good speed-ups for smaller inputs* by comparing two hypothetical parallel computer systems, called A and B. The only difference is that System B allows speed-ups for smaller inputs than System A. *Assumptions Concerning System A (resp. B):* In 95% (resp. 99%) of the time in which the list ranking routine is used speed-ups relative to the serial algorithm is achieved; the speed-up is by a factor of 100. The improvement from System A to B is significant! If a fraction F of computation can be enhanced by a (speed-up) factor of S, Amdahl's Law ([HP96]) implies overall speed-up of $\frac{1}{(1-F)+F/S}$. So, overall speed-up for system A is $\frac{1}{0.05+.95/100} = 1/0.0595 \leq 17$ and for system B ≥ 50; nearly 3 times the speed-up for System A! (ii) *Choosing a fastest list* *ranking algorithm is more involved than implied by the literature.* A challenge for our experimental work has been to compare some new insights we have with the known experimental work on list ranking. While our results affirm that the RM (Reid-Miller) algorithm is faster for large inputs, for smaller inputs a new variant of [Vi84], called the No-Cut algorithm, offers competitive performance. (iii) *Analytic (non-asymptotic) performance analysis is possible for smaller inputs.*

Due to space limitations, we must refer interested readers to [DV99].

2 List-Ranking Algorithms

The input is an array of N elements, each having a pointer to its successor in a linked list (the successor can be anywhere in the array). The pointer is labeled with the distance it represents. The *list ranking problem* is to find for each element in the list its total (weighted) distance from the end of the list.

To evaluate our list-ranking algorithms, we implemented a serial list-ranking algorithm, with serial assembly code, and several parallel list-ranking algorithms with our new explicitly parallel assembly code. For each of the codes, we aimed at figuring out the best ILP performance, taking into account constant factors and not only asymptotic behavior. The serial algorithm is followed below by three parallel algorithms: No-Cut, Cut-6, and Wyllie's algorithm. See Section 7.1 for RM's "random subset" algorithm.

Serial Algorithm The serial algorithm for the problem consists of first finding the "head" of the list; the head is the only element of the list that is not the successor of any other element in the list. Second ("forward loop"), the list is traversed to find the ranking of the "head", determining for each element by how much closer it is to the "tail" of the list than the head. Finally ("final ranking"), the ranking of all elements in the list are derived.

Parallel Randomized Coin-Tossing Algorithm (No-Cut) The main new parallel algorithm we used is randomized and works in iterations. See Figure 1. Assuming an array with precomputed results of a "randomly tossed coin" (0 or 1) is provided, an iteration first assigns such 0 or 1 to each element of the list. Every element, which is assigned 1 whose successor is assigned 0, is "selected"; other elements are not selected. Note that: (i) on average a quarter of the elements are selected, and (ii) it is impossible to select two successive elements. So far this is the same basic idea introduced in [Vi84]. The new part is: for each selected element, "pointer-jump over" its successor elements, stopping once another selected element is reached. This results in a new list which contains only the selected elements. (*In the next section the length of the "chain" between selected elements is observed to be relatively short, implying that the performance of this "No-Cut" algorithm is attractive.*) The selected element, which pointer-jumped over a non-selected element, is its "ranking parent". All selected elements are included in a smaller array. The compaction into the smaller array is achieved using a new multiple-PS instruction. The smaller array is the input for the next iteration. These "forward iterations" will result in finding the rank of the head of the list. All the elements which have been jumped over (i.e., skipped) during each forward iteration are compacted into another smaller array noting for each its "ranking parent". Playing "backwards" the iterations will extend the ranking from ranking parents to all elements of the list. Our evaluations show that for some input sizes this No-Cut algorithm requires less work (but may need more time) than other versions and algorithms.

Time-Emphasized Parallel Randomized Coin-Tossing Algorithm (Cut-6) A variant of the parallel algorithm of the previous paragraph is presented next. Dubbed "Cut-6", it achieves better time but does more work. That is, it runs faster than the No-Cut algorithm if available (machine) parallelism is above some threshold. The only difference relative to the No-Cut algorithm is that in the forward iterations a selected element stops pointer-jumping after 6 elements even if it has not reached another selected element. Forward iterations also find the rank of the head of the list and playing the iterations "backwards" extends the ranking to all elements of the list.

Wyllie's Parallel Algorithm consists of $log_2 N$ steps, each comprising parallel pointer jumping by all elements, till all elements are ranked.

3 Optimization and Analysis of the Algorithms

The Level of Parallelism denoted P is (for an ILP architecture) *the number of instructions whose execution can overlap in time.* P cannot exceed the product of the number of instructions that can be issued in one clock with the number of stages in the execution pipeline. (The popular text [HP96] suggests in p. 430 that multiple-issue of 64 is realizable in the time frame of the years 2000-2004; so, a standard 5- or 6-stage pipeline leads to P exceeding 300. It is hard to guess why P has hardly increased since 1995; perhaps, this is because vendors have not found a way to harness such an increase for cost effective performance improvement of existing code.)

Our building blocks for parallel list ranking are several parallel algorithms each of which is *non-dominated*, i.e., for each of these algorithms there is some input size and some P value for which it is the fastest. [DV99] describes low- and high-level considerations that were involved in getting the best performance out of each algorithm, and for assembling them in line with the accelerating cascades technique (see [CV86a]). This section discusses particularly noteworthy issues related to work and time analysis of the algorithms.

Serial Algorithm
Loop unrolling is the main concern in analyzing the execution of the serial algorithm. Issues such as the size of the code, the limited number of registers R, and the overhead of leaving too large a "modulo" from the loop, are considered. Most limiting was the value of R, and [DV99] discusses a choice of $R = 128$.

Parallel Randomized Coin-Tossing Algorithm (No-Cut)
Two issues require most attention: (i) the convergence rate of the forward iterations, and (ii) bounding the length of the *longest chain* from one selected element to its successive selected element. The No-Cut algorithm is optimized for less work. It has smaller work constants than the time-emphasized Cut-6 algorithm, but its asymptotic time bound includes a $O({log_2}^2 N)$ term. The algorithm becomes dominant (in the term $\frac{W}{P} + T$ - a standard way for evaluating PRAM algorithms) for relatively small values of P. The work term is $O(N)$ and its constant is very important. *But, why does the "No-cut" algorithm perform less work?* The answer is: a significant decrease in the number of (forward and backward) iterations. We have longer threads but fewer of them in fewer iterations, so on balance the overall work term is smaller. (An optimization not discussed here is for reducing the number of spawn-join pairs.)

Convergence of Iterations The expected fraction of skipped elements in each iteration is $\frac{3}{4}$. Let N be the initial input size, and let M_i be the expected number of active elements in the i-th iteration (i.e., the input size for the i-th iteration); we make the simplifying assumption that the number of active elements in each iteration will be: $M_0 = N, M_1 \le \frac{1}{4}N, \cdots, M_i \le (\frac{1}{4})^i N$. So, $\Sigma M_i \le 1.34N$. The number of forward iterations $\#iter$ will not exceed $log_4 N = 0.5 log_2 N$.

According to the simulations for various sizes of inputs, $\Sigma M_i \le 1.33N$ and $\#iter \le 0.46 log_2 N$; both fully supporting our analysis.

Bounding the Length of the Longest Chain In order to analyze the execution time for the *No-Cut* algorithm we need to find a realistic bound on the longest chain. This bound is important since the critical-time-path length of each forward iteration is determined by the "slowest" thread, i.e., the thread that has to short-cut over the longest chain.

Some definitions follow: (i) A *"chain"* is a series of consecutive (based on pointers) elements in a list starting at "1,0" (coin-tossing selection results) and ending just before the next "1,0" (coin-tossing selection results). (ii) A *chain's "length"* is the number of elements *after* the initial "1" (i.e., the elements which will actually be short-cut). (iii) Given an element, $P(length \ge i)$ is the probability that the length of a chain starting at it is i, or longer.

Claim 1: $P(length \ge logN + loglogN) \le \frac{1}{N}$.
Verbally, this can be described as: the probability for chains whose length is longer than $logN + loglogN$ is upper bounded by $\frac{1}{N}$.

To prove Claim 1, we find an i such that $P(length \ge i) \le \frac{1}{N}$. *The probability of the length of a chain* will be determined as follows: (i) A chain will be of length $\ge l$ if $\forall i : 1 \le i < l, next^i(x) is\ not\ selected$ where: (1) x is the selected element that starts the chain, and is considered to be the zero element of the chain. (2) $next^i(x)$ is the i-th successor of element x. And, (3) an element is selected only if its coin is "1" and its successor's coin is "0". (ii) $\frac{1}{4}$ of the $next^2(x)$ elements will be selected (when $next^2(x) = 1$ and $next^3(x) = 0$). $\frac{2}{8}$ of the $next^3(x)$ elements will be selected, since only 2 out of their successors which are still active (8 options) will be $next^4(x) = 0$. (iii) So the series of selected elements will be: $0, \frac{1}{4}, \frac{2}{8}, \frac{3}{16}, \frac{4}{32}, \frac{5}{64}, \ldots, \frac{i-1}{2^i}, \ldots$. The situation is depicted in Figure 2. For the algorithm this means the following: the fraction of threads that will stop after 1 short-cut is $\frac{1}{4}$, after 2 short-cuts is $\frac{2}{8}$, after 3 short-cuts is $\frac{3}{16}$, and so on.

Claim 1 follows from $P(i) = \frac{i+1}{2^{i+2}} \leq \frac{1}{n}$.

[DV99] reports several simulations to support our formal analysis of the longest chain bounds. The simulations resulted in the following: (i) We ran over 200 different N-values in the range: $10^2 \leq N \leq 10^7$. (ii) $Maxlen < 1.23 * log_2 N$; $Maxlen$ denotes the length of the longest chain. (iii) Only 6 simulations resulted in $Maxlen \geq 1.20 * log_2 N$. (iv) Practical conclusion: $E(Maxlen) = 1.2 * log_2 N$.

The Cut-6 Algorithm For analyzing the Cut-6 algorithm we consider the sequence which describes the convergence of the forward iterations. Recall that during the forward iterations, a selected element stops pointer-jumping after 6 elements even if it has not reached another selected element. An analysis which is similar to the proof of Claim 1 applies. The expected fraction of skipped elements in each iteration will be $[\frac{1}{4} + \frac{3}{16} + \frac{4}{32} + \frac{5}{64} + \frac{6}{128} + \frac{7}{256}] = \frac{183}{256}$. Let N be the initial input size, and let M_i be the number of active elements in the i-th iteration. Then, under similar assumptions to the analysis of the no-cut algorithm, the number of active elements in each iteration will be: $M_0 = N, M_1 \leq \frac{73}{256}N, \cdots, M_i \leq (\frac{73}{256})^i N$ and $\Sigma M_i \leq \frac{256}{183}N \leq 1.4N$. The number of iterations necessary for the algorithm to finish is $log_{(256/73)}N = 0.552 log_2 N$.
Why, among the family of cut-k algorithms, did we chose the one that stops after short-cutting at most 6 elements? We experimented with $k = 3, 4, 5, 6, ..., 11$. The minimum time constant, for the $O(log_2 N)$ term, was obtained for $k = 6$. The work constant, for the $O(N)$ term, continued decreasing until $k = 10$, but wasn't significantly smaller than for $k = 6$. Thus we chose k to be 6.

Wyllie's Parallel Algorithm is relatively simple. Most of its work and time analysis is clear from the code and the counts presented. The only place where additional assumptions were needed concerns the calculation of $log_2 N$ (for the number of iterations). As discussed in [DV99], the $log_2 N$ calculation adds a large constant to the work term.

General comment on compilation feasibility Translation from the high level code (C and extended-C) to the optimized assembly code, which has actually been done manually, is feasible using known compiler techniques. See [DV99].

4 The Instruction Set

The parallel algorithms presented in this work all use an asynchronous mode of the so-called *Spawn-MT programming model* for the XMT framework; this allows expressing the fine-grained parallelism in the algorithms. Concretely, we extended standard instruction sets (for assembly languages) to enable transition back and forth from serial state to parallel state and allow in the parallel state any number of (virtual) threads. A Spawn command marks a transition from a serial state to parallel state and a Join command marks a transition in the opposite direction. See left side of Figure 3. The semantics of the assembly language, called the *"independence of order semantics (IOS)"*, allows each thread to progress at its own speed from its initiating Spawn command to its terminating Join command, without having to ever wait for other threads. Synchronization occurs only at the Join command; all threads must terminate at the Join command before the execution can proceed.

The assembly language includes *multi-operand* instructions, mainly the **prefix-sum** (PS) instruction, whose specification is similar to the known Fetch-and-Add. The paper [Vi97] explains why if the number of operands does not exceed k, for some parameter k which depends on the hardware, and each of the elements is, say, one bit, multiple-PS can be implemented (in hardware) very efficiently (in "unit time"), even where the base is any integer. The **high-level language** for Spawn-MT (as well as the assembly language) is single-program multiple-data (SPMD). It is an extension of standard C which includes Spawn and multioperand instructions such as prefix-sum. Using standard C scoping, the Joins are implicit. A Join implementation comprises a parallel sum operation for monitoring the number of terminating threads.

The XMT instruction set extends the standard serial MIPS instruction set (see [HP94]). The table describes a few new instructions. A fuller specification for the explicit parallel instruction set (also referred to as spawn-join instruction set) can be found in the extended summary version of [VDBN98]; it describes the new instructions and how they can be efficiently implemented in hardware.

Few XMT Instructions

Notation: $R1\$$ is local register number 1 of thread $\$$.

Instruction	*Instruction Meaning*
$PS\ Ri, Rj$	Atomically: (i) $Ri := Ri + Rj$, and (ii) $Rj := (original) Ri$. Ri called *base for PS.*
$PS\ Ri, Rj_f$ for $1 \leq f \leq g$	Multiple prefix-sum: If a sequence of g Prefix-sum commands comes from one thread: follows serial semantics. If the g commands come from different threads: follow some (arbitrary) serial semantics. Execute by special hardware. Assumptions made: if each Rj_f between 0 and 3, then unit time execution if $g \leq k$ (base is any integer).
$PSM\ 15(Ri)Rj$	*Prefix sum to memory*, with base $Memory[15 + Ri]$. Multiple PSM: only inter-thread.
$Mark\ Ri, Rj$	If $Ri = 0$, then atomically: $Ri = 1$ and $Rj = 0$; else $Rj = 1$. Also: Multiple-Mark executed by special hardware.
$MarkM\ 15(Ri), Rj$	Mark to memory. Also, multiple MarkM.

5 Execution Model & Execution Times

We are interested in comparing performance results for the serial algorithm in a serial (ILP) execution model with the parallel algorithms in our new execution model. The performance results we obtained are derived by finding the execution critical-time-path T and counting the work W from the assembly code written for each of the algorithms in their respective instruction sets.

Following is an overview of the process by which we obtained the performance results.
(i) Given the code, we tried to place each instruction as early as possible without violating data and control dependences.
(ii) *Speculation.* Generally, we were able to obtain our results without resorting to speculation; however, we allowed speculation for branches which are known to happen with high probability (e.g., branching on whether an element of a linked list is last in the list fails for all but one of the elements of the list).
(iii) *Cache misses.* We examined every memory access. If, due to spatial-locality or temporal-locality, the address could be found in the cache, we assumed unit-time memory access. If a case for spatial or temporal locality could not be made, we applied a penalty of $d = 5$ time units (to simplify our analysis).
This is unquestionably an arbitrary decision which oversimplifies an involved situation. The issue is how to account for memory response time without resorting to detailed memory system simulations; the problem is that to reach the level of detail that merits the use of simulations, we will have to specify the hardware in ways which exceed our more general bridging model and our basic intent to reserve judgement on hardware implementation issues which complement our new ideas. We could only say that picking somewhat bigger or smaller numbers for d did not appear to change our overall comparison of serial and parallel code significantly. For problems, which are not exceedingly large, the memory could fit within the levels of the cache (which reached 16MB for all cache levels in 1995, with a maximum response time of 15 cycle, see [HP96]). So, the number 5 is not unreasonable for average response time. Also: (a) the sensitivity of our speed-up results to replacing 5 by slightly bigger or smaller numbers was marginal; (b) presented and explained in [VDBN98], a Scheduling Lemma justifies the use of a single average number for simulating memory response time.
(iv) *Prefetching.* To alleviate this cache-miss penalty, prefetching could begin prior to a branch or spawn instruction, that precedes the memory access; however, to minimize speculative execution, their commit is deferred to after the resolution of the branch or spawn.
(v) *Spawn and Join instructions.* Our analysis uses the following simplifying assumption: a Spawn instruction contributes one to the operation count, and a Join instruction (in each thread) is counted as one operation. Our rationale follows. The principal goal has been comparing the performance of two kinds of instruction codes: serial and parallel. We were faced with a situation where the extensive hardware used currently in practice in order to extract ILP from serial code and control its execution is not accounted for in the instruction code. This gives an unfair advantage to the serial code since it is likely to require more complicated hardware than for implementing our explicitly parallel code; one reason is that *parallelism mandated by the code frees the hardware from verifying that no dependences occur.* This stresses our case. It explains why in terms of devoted hardware, standard serial code is unlikely to be implemented faster, or using less

hardware, than our envisioned parallel code (for the same level of parallelism). Still, we take the extremely conservative approach of counting the Spawn and Join instructions, while essentially not counting anything on the serial code end. The only case where we applied to serial code the same costs as to the parallel one was where loop parallelism, which can be determined at run-time, is needed. Note that this still reflects a "discount" for the serial code: in practice only some fixed loop-unrolling is possible for serial code, while our Spawn mechanism has no problem handling this for parallel code.
(vi) *Unit time instructions.* All other instructions are assumed to take unit time.
(vii) *Scheduling Lemma, number of registers and code length.* The Scheduling Lemma (in [VDBN98]) assumes that the number of registers is unlimited and that the length of the code can be increased with no penalty (as needed for loop-unrolling). (However, our performance results consider limiting these assumptions, as well.)

The detailed execution assumptions and techniques for figuring out time and work counts are described under `http://www.umiacs.umd.edu/~vishkin/XMT/` in *"Explicit Multi-Threading (XMT) - Specifications and Assumptions"*; they are followed there by a simple example, to facilitate understanding before approaching the full algorithms. The proof of the Scheduling Lemma is presented in the full paper of [VDBN98].

A Conservative Comparison Approach We made every effort to avoid unfairly favoring the performance estimates of the new code over standard serial code. Concretely, we followed the following guidelines:

(1) For standard serial code, we took T as the running time, since this is a **lower bound** on the execution time achievable.
(2) For parallel code, we took $W/p+T$ as the running time, since this is an **upper bound** on the execution time achievable. (This bound relies on the Scheduling Lemma in [VDBN98].)

6 Summary of Work and Time Execution Counts and Speed-up Results

The execution times for the four algorithms were derived by counting the execution time and work of each algorithm, following the execution assumptions and techniques described above. The fully commented assembly code for each of the algorithms can be found through the web site `http://www.umiacs.umd.edu/~vishkin/XMT/`. Detailed counts are also presented there in a table format for each of the algorithms, and the execution work and time below are derived from that. (For later reference, we also included data on the RM algorithm.)

Algorithm	Work $\leq$	Time
Serial[2]	$15N + 4(\frac{N}{R}) + 5(N mod R) + 22$	$9N + 4(\frac{N}{R}) + 4(N mod R) + 22$
No-Cut[3]	$58N + 4log_2 N + 2$	$4.2{log_2}^2 N + 18.5 log_2 N + 27$
Cut-6	$80.5N + 2.2log_2 N + 2$	$44.7 log_2 N + 27$
Wyllie[4]	$10N log_2 N + 12N + 2log_2 N + 268$	$15 log_2 N + 30$
RM[5]	$32N + 10 log_2 N + 9$	$9{log_2}^2 N + 27 log_2 N + 6$

6.1 Speed-up Results

We compared the execution time for the 4 algorithms (serial, No-Cut, Cut-6 and Wyllie's) for various input-size (N) values and machine parallelism (P).

We considered P values up to 2000, which cover the horizon for the amount of parallelism currently envisioned for explicit ILP.

We considered N values (input size) up to 10^7.

Upper bounds on the execution time for the parallel algorithms ($\frac{W}{P} + T$) are compared with a lower bound on execution time for the serial algorithm (T), to comply with our conservative approach.

We noticed the following: (i) The serial algorithm, under our assumptions and execution counts, is competitive for $P < 7$. (ii) Each of the three parallel algorithms has an area (of P-values and N-values) for which it is the best (see table at the end). When looking at a specific P (along a vertical section) we see, as expected, that:

2 In the serial algorithm: (i) The loop unrolling is bound by R, the number of available registers (which affects the work and time counts). (ii) For the comparison with the parallel algorithms we will only look at the *lower bound* of the serial algorithm, i.e., the time count alone (which amounts to a critical path in a dependence graph).

3 The ${log_2}^2 N$ term in the time count for the "work-emphasized" No-Cut algorithm is based on the fact that the "longest chain" (in each iteration of the No-Cut algorithm) does not exceed $2 log_2 N$ with high probability.

4 The high constant in Wyllie's algorithm's work is due to the calculation of $log_2 N$ (the number of iterations) at the beginning (assuming the N is 64 bits).

5 In the RM algorithm: (i) we used a powerful atomic instruction to eliminate multiple selections of the same element (and remain with one copy of each selected element). An alternative setting, where such an atomic instruction is not available, would imply that the the time and work counts increase. (ii) The random selection of elements is counted as one operation per iteration only. This implies that the random numbers should be prepared ahead of time and at a minimum scaled (which requires at least one additional operation) to the relevant list size at each iteration. These assumptions provide wide margins to make sure that we do not favor the new algorithms (No-Cut, Cut-6, etc) to RM's algorithm. As a result the work and time counts for RM's algorithms ended up being a bit optimistic. See later reference in Section 7.1.

- For larger N-values, the work-emphasized No-Cut algorithm is ahead.
- For intermediate N-values, the time-emphasized Cut-6 is ahead.
- For smaller N-values, Wyllie's algorithm is ahead.

Plots of the areas in which each algorithm is dominant are shown in Figure 4. Note that the table at the end of the paper adds data about RM's algorithm and shows that it is ahead for larger N-Values.

7 Other Implementations of List-Ranking Algorithms

In [Re94] list ranking is discussed. After considering several work efficient list ranking algorithms, such as [MR85], [AM90] and [CV89], they implemented a new algorithm which has experimentally optimized smaller constants for the parallel work, but not fast asymptotic time. The algorithm does recursive iterations using a randomized division into $m+1$ sublists, where $m < \frac{n}{logn}$. The variance in the size of the sublists (which does not present a problem in our XMT model) is amortized by giving several sublists to each processor. When the number of sublists becomes small enough (determined experimentally) they switch to use either the serial or Wyllie's algorithm. Note that the code requires a careful manual adjusting of parameters to achieve good speedup each time a different number of processors is used. The platform used is the *Cray C90* vector multiprocessor, with 128 vector elements per processor (in Figure 2 there). As explained there, when vectorizing a serial problem that requires gather/scatter operations, the best speedup one can expect on a single processor *Cray C90* is about a factor of 12-18, but when the vectorized algorithm does more work than the serial one then one can expect smaller speedups (the vectorization is used to hide latencies). The results presented are for 1, 2, 4 and 8 processors. The speedups obtained for list ranking (presented in Figure 11 there) using 8 processors (i.e., $8*128 = 1024$ vector elements) are approximately 40 for $n = 3*10^7$, 35 for $n = 4*10^6$ and 4 for $n = 8*10^3$. In Section 7.1 below, a detailed comparison between RM's algorithm and our No-Cut algorithm is given.

In [HR96] the implementation of three parallel list ranking algorithms on the massively parallel SIMD (Single Instruction Multiple Data) machine *MasPar MP-1* with virtual processing, using the MPL C-based language is presented. The *MP-1* has 16384 physical processors. The algorithms implemented were: Wyllie's suboptimal deterministic algorithm, a simple optimal randomized algorithm and a combination of the two where initially the randomized algorithm is run and when the number of elements remaining was no more than the number of physical processors they switch to Wyllie's algorithm. A comparison to the sequential algorithm on a *UNIX* system is presented, where the raw computational power of the *MP-1* was at least 63 times larger than the *SPARC* used, and the total amount of main memory available for computation on the massively parallel computer is approximately 32 times larger. A speedup of 2 was achieved when all cells in the linked list fit into the *SPARC*'s main memory ($n \simeq 3*10^6$). However, when heavy swapping was needed in the sequential implementation, its performance degraded dramatically.

The platform used in [Si97a], [Si97b] and [SGS] is the *Intel Paragon*, which has a grid like structure. Three algorithms were implemented in [SGS]: Wyllie's pointer-jumping, independent set removal and sparse ruling set. Using 100 processors the best speedups were respectively: 5 for $n \simeq 3*10^6$, 14 for $n \simeq 5*10^7$ and 27 for $n \simeq 2*10^8$. In [Si97b] the main result is an "external algorithm" which achieves a speedup of 25 using 100 processors for $n = 10^8$; for an illustration of the advantage of XMT over these multi-processing results for smaller input, see right side of Figure 3.

Several deterministic work-optimal EREW list ranking algorithms (see [Tr96]) have been implemented using *Fork95* for the PAD library of PRAM algorithms and data structures in connection with the the *SB-PRAM* and *Fork95* projects. *Fork95* is a c-based parallel programming language. All the algorithms switch at some experimentally determined parameter to Wyllie's algorithm. The execution was evaluated on a simulator for the SB-PRAM. The results reported were: with 32 processors for $n = 42968$ there is a break-even between Anderson-Miller's algorithm and the serial one; with 128 processors the best algorithm is Wyllie's with speedups of approximately 3.5 for $n = 42968$ and 5.7 for $n = 10742$. Their practical conclusion was that none of the studied algorithms is fully satisfactory, due to large constants; although some randomized algorithm might achieve better results.

7.1 Detailed Comparison to Reid-Miller's Algorithm

RM's algorithm also has forward and backward iterations. A forward iteration works as follows. A parameter k is specified. In parallel, k random numbers each between 1 and n (the number of current elements in the list) are picked. Each element, whose index is one of these random numbers, is selected (after getting rid of multiple selections of the same element). For each selected element the forward iteration proceeds by pointer jumping over all non-selected elements. This results in linking all selected elements. All selected elements are then included in a smaller array. RM's algorithm performs less work at one interesting point. Unlike the No-Cut and Cut-6 algorithms it need not check for each element (at the beginning of the forward iteration) whether it was selected. We implemented RM's algorithm in the same XMT programming model. Since our main interest was comparison with our No-Cut algorithm, we implemented RM's algorithm in assembly code for the XMT computational framework. (We left the cascading aside as we did for other algorithms we implemented.)

For comparison purposes, we took the number of randomly picked sublists to be $k = \frac{n}{4}$. Note, however, that in the No-Cut algorithm, we could have actually controlled the expected number of selected elements by picking a probability different than $\frac{1}{2}$ for getting a Head at each node. In other words, No-Cut can also be flexible with the number of selected elements similar to RM's algorithm.

We note several things in our implementation of RM's algorithm and in the comparison to the No-Cut algorithm. First, the random selection of k elements, as the heads of the sublists, is presented and counted as only one operation per iteration in the instruction code. This implies that the random numbers should be prepared ahead of time and (at least) scaled (which should require at least one additional operation for each draw of a number in each iteration) to the relevant list-size at each iteration (to achieve uniform distribution for each iteration). Second, we used the `markM` atomic instruction (which is a simpler variant of the PSM instruction) to eliminate multiplicities among the k selected elements. This instruction is counted as 1 work unit and 5 time units (since it has to read a memory location). Had multiplicities been eliminated differently the time and work necessary per iteration would have been different. These assumptions were used as part of a conservative comparison with RM's algorithm. We expected RM's algorithm to perform less work, since only the threads that are active (i.e., selected) start working after the selection was completed. In the No-Cut implementation all threads are spawned and those who aren't selected terminate immediately (which accounts for some of the additional work). We should remember that the No-Cut algorithm was time-optimized to compete with the Cut-6 algorithm, thus putting most of the effort into decreasing the critical-time-path even when that results in additional work. This included, amongst other things, using an "active-array" and copying into it most of the information in the elements, in order to save a few cache misses (when accessing data through two pointer accesses, which must be done in RM's algorithm). When optimizing for less work a different approach should be used for a better comparison. In such a case, we could save roughly $15 * N$ work units by using a selected-list (similar to the one used in RM's algorithm) at the end of the selection step (coin-tossing), spawning only the selected elements and not storing duplicates of most of the information in an active-array. This, of course, would increase the critical-time-path (by at least $5 * log_2 N$), and both algorithms would be much more similar.

The discussion above explains how our performance results made RM's algorithm look much better than other algorithms. This was done since the comparison was against RM's algorithm. The results of the analysis show that the work term for the RM algorithm ($32N + 10log_2 N + 9$) is smaller than the one for the No-Cut algorithm (see the results in section 6), but the time term for the No-Cut algorithm is smaller than the one for RM's algorithm ($9log_2 N^2 + 27log_2 N + 6$). For smaller list sizes the No-Cut algorithm performs better. A few of the speedups, comparing to the serial algorithm, obtained for RM's list ranking and for the No-Cut algorithm on the XMT are compared to the ones on the Cray-90 in the following table.

Input Size n	RM on Cray-C90 $P = 8 * 128 = 1024$	RM on XMT $P = 1000$	No-Cut on XMT $P = 1000$
$3.5 * 10^7$	$speedup \simeq 40$	$speedup \simeq 280$	$speedup \simeq 155$
$4.2 * 10^6$	$speedup \simeq 35$	$speedup \simeq 270$	$speedup \simeq 154$
$8 * 10^3$	$speedup \simeq 4$	$speedup \simeq 34$	$speedup \simeq 51$

Due to the new mechanism for enhancing thread spawning using PS (as per [VDBN98]), XMT circumvents the main hurdle for implementing RM's algorithm on the Cray, which is the need to group several sublists for amortizing unequal distributions of length. RM's solution used larger input length to over-come this hurdle. However, this, in turn, increased the input length for which good speed-ups are obtained. For XMT one can get good speed-ups already from shorter inputs. We should also remember that we favored the RM implementation on XMT for the comparison with No-Cut; this in turn somewhat biases its comparison to the implementation on the Cray. Detailed speedup results are presented in [DV99]. The analysis and simulations presented there are summarized below. The *most interesting conclusion* in this analysis is that the expected length of the longest chain in RM's algorithm is $\simeq 2.75 * log_2 N$, while for the No-Cut algorithm Section 3 derived the result $E(Maxlen) \simeq 1.2 * log_2 N$. This highlights an interesting advantage of the No-Cut algorithm; this advantage is due to its different randomized mechanism for picking "selected" elements. This explains why the time term for RM in Section 6 is larger than for No-Cut and why No-Cut is ahead for certain input sizes and values of P.

Few more details on the analysis of the RM algorithm A "paper and pencil" analysis of the behavior of RM's algorithm was done picking $k = \frac{n}{4}$ elements randomly at each iteration as heads of the sub-lists. Also, results obtained from simulations of the algorithm's behavior using the same k are reported.

We wish to find the probability that an element is selected; i.e., picked at least once during the $k = \frac{n}{4}$ draws. The probability that a specific element is *not* selected at all during all the draws, denoted as p_{ns}, is:

$$(1 - \frac{1}{n})^{\frac{n}{4}} = ((1 - \frac{1}{n})^n)^{\frac{1}{4}} \xrightarrow[n \to \infty]{} (\frac{1}{e})^{\frac{1}{4}} = 0.779$$

Notice that the limit of $(1 - \frac{1}{n})^n$ goes to $\frac{1}{e}$ very quickly (it is already very close, 0.366 from 0.3679, for $n = 100$). And the probability that an element is selected is $(1 - 0.779) = 0.221$.

This led to an analytical bound of $2.77 * log_2 N$ on the length of the longest chain. This was backed by experimental results whose practical conclusion was that $E(Maxlen) \simeq 2.75 * log_2 N$.

References

[AM90] Anderson R. and Miller G.L. A Simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5), 1990, 269-273.

[CV86a] Cole R. and Vishkin U. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proc. 18th Annual ACM Symp. on Theory of Computing*, 1986, 206-219.

[CV89] Cole R. and Vishkin U. Faster optimal parallel prefix sums and list ranking. *Information and Computation* 81(3), 1989, 334-352.

[DL99] W.J. Dally and S. Lacy. VLSI Architecture: Past, Present, and Future, Proc. Advanced Research in VLSI Conference, Atlanta, March 1999.

[DV99] S. Dascal and U. Vishkin. Experiments with list ranking for Explicit Multi-Threaded (XMT) instruction parallelism (extended summary), UMIACS-TR-99-33, June 1999.

[HP94] Hennessy J.L., Patterson D.A., *Computer Organization and Design - The Hardware / Software Interface.* Morgan/Kaufmann. 1994.

[HP96] Hennessy J.L., Patterson D.A., *Computer Architecture - A Quantitative Approach*, 2nd Edition. Morgan/Kaufmann. 1996.

[HR96] Hsu T. and Ramachandran V. Efficient massively parallel implementation of some combinatorial problems. *Theor. Comp. Sci.* 6(2), 1996, 297-322.

[MR85] Miller G.L. and Reif J.H. Parallel tree contraction and its applications. In *Proc. Symposium on Foundations of Computer Science*, 1985, 478-489.

[Re94] Reid-Miller M. List ranking and list scan on the Cray C90. In *Proc. 6th ACM-SPAA*, 1994, 104-113.

[Si97a] Sibeyn J.F. Better Trade-Offs for parallel list ranking. In *Proc. 9th ACM-SPAA*, 1997, 221-230.

[Si97b] Sibeyn J.F. From Parallel to external list ranking. *TR-MPI-I-97-1-021*, Max-Planck-Institut Fur Informatik, Saarbrucken, Germany, 1997.

[SGS] Sibeyn J.F., Guillaume F. and Seidel T. Practical parallel list ranking. *Preprint*, 1997.

[Tr96] Traff J.L. List ranking on the PRAM. *The PAD Library Documentation*, Max-Planck-Institut fur Informatik, Saarbrucken, 1996.

[Vi84] Vishkin U. Randomized speed-ups in parallel computation. In *Proc. 16th ACM-STOC*, 1984, 230-239.

[Vi97] Vishkin U. From algorithm parallelism to instruction-level parallelism: An encode-decode chain using prefix-sum. In *Proc. 9th ACM-SPAA*, 1997, 260-271.

[VDBN98] Vishkin U. et al. Explicit Multi-Threading (XMT) bridging models for instruction parallelism (extended abstract). In *Proc. SPAA'98*, 1998. For greater detail see the extended summary of this extended abstract under `http://www.umiacs.umd.edu/~vishkin/XMT/` .

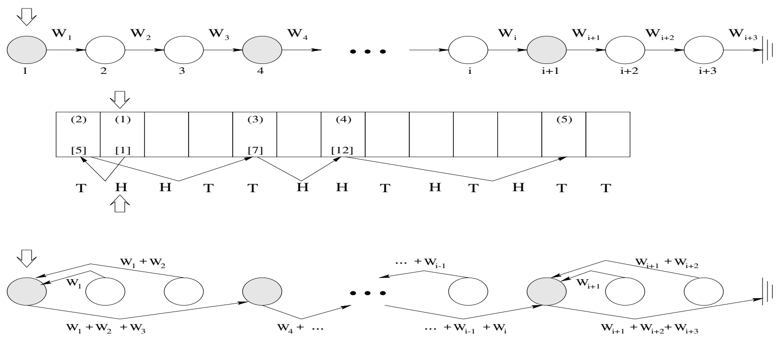

Fig. 1. *Upper figure:* original linked list. $\Downarrow$ - head of linked list. W_i - weight of a link. Elements numbering - their order in the linked list. *Middle figure:* input array representation of the linked list. Element numbering is in round brackets; pointer to array index of the next element in square brackets. Selected elements (H followed by T) are shaded. *Lower figure:* the linked list after one forward iteration. Backwards arrow to ranking parent.

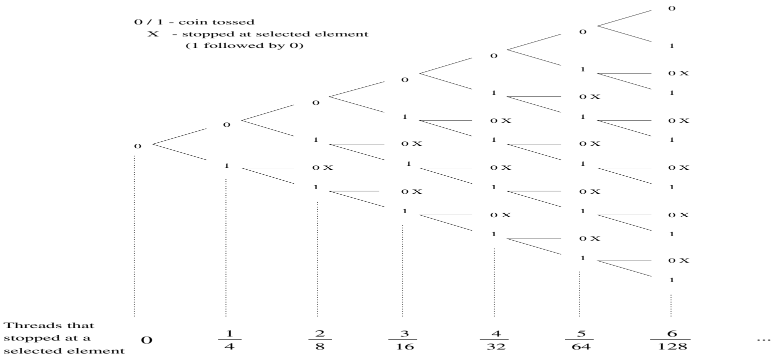

Fig. 2. Probabilities of selected elements and of chain length

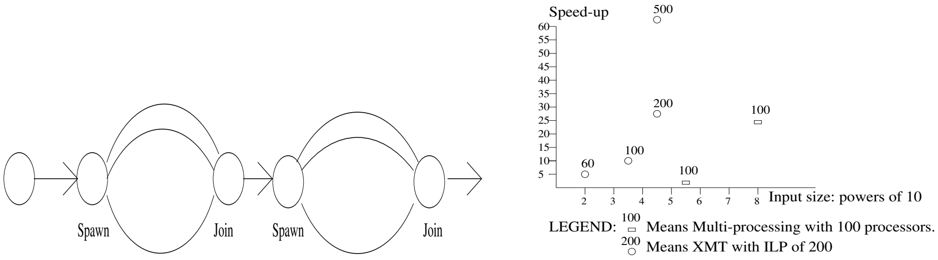

Fig. 3. Left figure: Parallel states from a Spawn to a Join and serial states. Right figure: Comparison to [Si97b].

N input size	P: 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.44	2.47	3.31	3.98	4.54	5.00	5.73	6.28	7.78	9.08	9.62	9.81	9.91
	alg 2	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4
10^3	1.47	2.68	3.69	4.55	5.29	5.93	6.99	7.83	12.81	23.47	32.49	37.26	40.22
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 4	alg 4	alg 4	alg 4	alg 4
$5 * 10^3$	1.51	2.94	4.28	5.55	6.76	7.90	10.02	11.93	19.33	32.73	50.86	68.08	81.95
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3	alg 4	alg 4	alg 4
10^4	1.53	3.01	4.44	5.83	7.17	8.46	10.94	13.26	23.09	41.57	63.39	78.09	94.77
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3	alg 3	alg 4
$5 * 10^4$	1.55	3.09	4.61	6.12	7.62	9.10	12.02	14.89	28.53	63.30	106.60	138.09	165.09
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3
10^5	1.55	3.10	4.64	6.17	7.69	9.20	12.21	15.18	29.62	68.99	123.85	168.53	205.62
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2
10^6	1.56	3.11	4.67	6.22	7.77	9.32	12.42	15.52	30.92	76.50	150.36	221.74	290.74
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2
10^7	1.56	3.11	4.67	6.23	7.78	9.34	12.45	15.56	31.11	77.67	154.98	231.93	308.52
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2

Table Above - Best Speedups Without RM's Algorithm These are the best speedups obtained when comparing the upper-bound on the parallel algorithms (not including RM's) with the lower-bound of the serial algorithm. *The algorithm used are: alg 1* - serial algorithm; *alg 2* - work optimized coin tossing algorithm (No-Cut); *alg 3* - time emphasized coin tossing algorithm (Cut-6); *alg 4* - wyllie's (pointer jumping) algorithm. Horizontal and vertical lines indicate a different algorithm on each side of the line.
Table Below - Best Speedups With RM's Algorithm Recall that the RM results (alg 5 below) are a bit optimistic, as explained in the paper.

N input size	P 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.46	2.47	3.31	3.98	4.54	5.00	5.73	6.28	7.78	9.08	9.62	9.81	9.91
	alg 5	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4
10^3	2.16	3.41	4.23	4.80	5.29	5.93	6.99	7.83	12.81	23.47	32.49	37.26	40.22
	alg 5	alg 5	alg 5	alg 5	alg 2	alg 2	alg 2	alg 2	alg 4	alg 4	alg 4	alg 4	alg 4
$5*10^3$	2.55	4.66	6.43	7.93	9.23	10.36	12.22	13.71	19.33	32.73	50.86	68.08	81.95
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 2	alg 3	alg 4	alg 4	alg 4
10^4	2.66	5.03	7.16	9.08	10.82	12.40	15.18	17.54	25.43	41.57	63.39	78.09	94.77
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 2	alg 3	alg 3	alg 4
$5*10^4$	2.78	5.47	8.08	10.60	13.05	15.43	19.98	24.27	42.55	77.64	107.07	138.09	165.09
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 2	alg 3
10^5	2.80	5.54	8.24	10.89	13.49	16.05	21.04	25.85	47.70	96.73	147.17	178.13	205.62
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 2
10^6	2.82	5.63	8.43	11.23	14.02	16.80	22.35	27.86	55.03	132.58	250.04	354.83	448.89
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5
10^7	2.82	5.64	8.46	11.28	14.10	16.92	22.55	28.17	56.25	139.91	277.46	412.70	545.69
	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5	alg 5

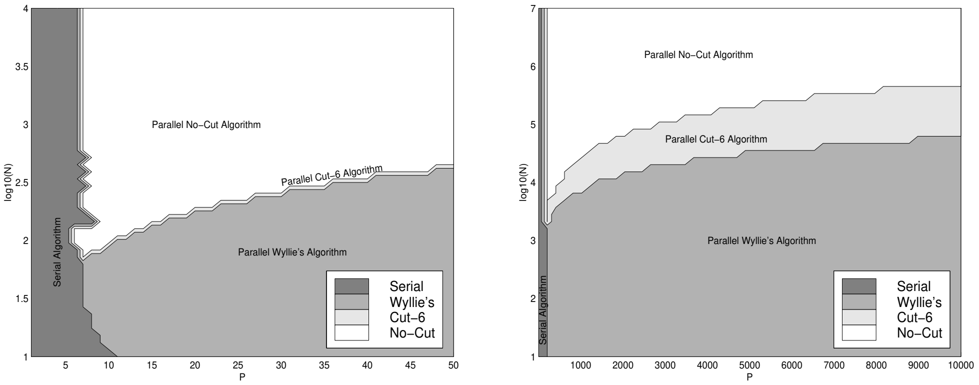

Fig. 4. The areas in which each of the four algorithms for list-ranking is dominant. Left figure: Parallelism up to 50. Right figure: Parallelism up to 10,000.