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

Shlomit Dascal Uzi Vishkin

June 1, 1999

Abstract

Algorithms for the problem of list ranking are empirically studied for the Explicit Multi-Threaded (XMT) platform for instruction parallelism. The main goal of this study is to understand the 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 [Vi84], 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. XMT covers the spectrum from algorithms through architecture to implementation; the main innovation in XMT (introduced in [VDBN98]) was through the incroporation 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 instruction-level parallelism.

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

[1] An extended abstract of this paper is due to be presented in the 3rd Workshop on Algorithm Engineering, London, U.K., July 1999. This footnote was added to explain why we chose this venue. 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. Instruction-level parallelism 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 this 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 real machine? Such comparisons are standard in computer architecture research; however, caution is indeed needed. (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? Since bandwidth within a chip is generally not a bottleneck (see, e.g., [DL99]) bandwidth does not explicitly appear.

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); 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. The XMT framework has been presented in [VDBN98]. (For those who are more familiar with the presentation there, we use in the current paper the so-called *Spawn-MT* programming model; this is the easier to implement among the two programming models presented in [VDBN98]). An overview of the relevant aspects of XMT is given in Appendix A. 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 9 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 reports of results for this problem which are competitive with the counterpart serial algorithm for input size not exceeding 1000, In fact, often the applicable input size 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. We do this using a hypothetical application for comparing two hypothetical parallel computer systems. The only difference is that System B allows speed-ups for smaller inputs than System A. *Assumptions Concerning System A:* In 95% of the time in which the list ranking routine is used speed-ups relative to the serial algorithm is achieved; the speed-up is a factor of 100. *Assumptions Concerning System B:* In 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 also a factor of 100. This appears like a modest improvement of 4%. However, if a fraction F of computation can be enhanced by a (speed-up) factor of S, Amdahl's Law (see, e.g., [HP96]) states that the overall speed-up will be $\frac{1}{(1-F)+F/S}$. So, Amdahl's Law implies that the overall speed-up for system A is $\frac{1}{0.05+.95/100} = 1/0.0595 \leq 17$. For system B, the overall speed-up is $\frac{1}{0.01+.99/100} = 1/0.0199 \geq 50$ nearly 3 times the speed-up for System A! (ii) *Choosing a fastest list ranking algorithm seems to be more involved than implied by the literature.* A challenging aspect of our experimental work has been to compare some new insights we have with the known experimental work on list ranking. While our results appear to affirm that the RM (Reid-Miller) algorithm is faster for large inputs, they show that 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.*

2. List-Ranking Algorithms

Suppose that the input is an array of N elements, each having a pointer to its successor in the 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. We aimed at figuring out the best 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. For RM's "random subset" algorithm, see Section 9.1.

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 it is closer to the "tail" of the list than its 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. Assume that an array with precomputed results of a "randomly tossed coin" is provided, and each iteration begins by assigning such a coin tossing result to each element of the list. The algorithm proceeds by "selecting" each element whose coin toss is 1 and whose successor's coin toss is 0; other elements are not selected. Note that: (i) on the 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]. However, the main progress in the computation is achieved through advancing to a new list which contains only the selected elements; this is done as follows; for each selected element, "pointer-jump over" its successor elements, stopping once a selected element is reached. (*In Section 4.2 the length of the "chain" between selected elements is observed to be relatively short, implying that the performance of the 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 the new multiple-PS instruction (see appendices A and B). 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 including an indication to the element's "ranking parent" (the final ranking of each element is then derived from its ranking parent's rank during "backward iterations"). Playing "backwards" the iterations will extend the ranking to all elements of the list. We chose this algorithm (dubbed "No-Cut") as a practical version of [Vi84]; our evaluations showed that for some range of input sizes this algorithm requires less work (but may need more time) than alternative versions and algorithms.

An illustration of the selection used in the algorithm and the linking of selected elements is depicted in Figure 1.

Time-Emphasized Parallel Randomized Coin-Tossing Algorithm (Cut-6) A variant of the parallel algorithm presented in the previous paragraph is presented next. It is dubbed "Cut-6". It achieves better time but does more work. It renders better results than the No-Cut algorithm if more (machine) parallelism is available.

The only difference relative to the No-Cut algorithm is that in the forward iterations a selected element performs pointer-jumping over at most 6 elements. These forward iterations will also result in finding the rank of the head of the list. Playing "backwards" the iterations will extend the ranking to all elements of the list.

The selection step used and the backward iterations are identical to those presented in the previous paragraph. Only the forward iterations are different.

Wyllie's Parallel Algorithm Another parallel algorithm we used for comparison with the serial algorithm is Wyllie's parallel algorithm. This algorithm consists of $log_2 N$ steps, each comprising parallel pointer jumping by all elements, till all elements are ranked (and pointing to the "tail" of the list).

RM's algorithm The RM algorithm is described in subsection 9.1 where we compare our new insights to it. As can be seen there the comparison is "conservative" penalizing the running times of the new algorithms, while "discounting" the XMT running time for RM's algorithm. This conservative comparison adds confidence to our claims that this list ranking algorithm offers the best performance *only* for large inputs.

3. Optimization Attempts and Considerations

One of our more interesting goals was to find a list-ranking algorithm that would be practical for relatively small input sizes and be competitive with the serial algorithm for low levels of parallelism. The level of parallelism is denoted by P. For an ILP architecture, P is the number of instructions whose exection 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 standard text [HP96] suggests in p. 430 that a multiple-issue of 64 is realizeable in the years 2000-2005 time frame; this implies that a 5- or 6-stage pipeline leads to P exceeding 300. It is hard to tell why P has hardly increased since 1995; perhaps because vendors have not found a way to harness such an increase for cost effective performance improvement of existing code.)

In line with the accelerating cascades technique (see [CV86a]) 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.

Let W_a and T_a be the work and time for algorithm A (No-Cut), W_b and T_b be the work and time for algorithm B (Cut-6), and W_c and T_c be the work and time for algorithm C (Wyllie's). We have a situation where $W_a < W_b < W_c$ but $T_a > T_b > T_c$. For concreteness see the table in section 8. In our execution model for the spawn-MT we look at a time upper bound, expressed by $\frac{W(N)}{P} + T(N)$. Each algorithm is the fastest for some N and P values, which implies that each of these three algorithms is indeed non-dominated.

Appendix F brings a description of some of the optimization considerations.

4. Algorithms Analysis

In this section we present more specific issues related to the analysis of the work and time for each of the algorithms.

4.1. Serial Algorithm

The main issue that arises when analyzing the execution of the serial algorithm is how to optimize it using *loop unrolling*. Several different considerations can be taken into account when unrolling a loop, such as the size of the code, the limited number of registers, the overhead of leaving too large a "modulo" from the loop, etc. Our main limit was the number of registers available.

Loop Unrolling For the serial list-ranking algorithm we unrolled the loop that does the final ranking. Since there are no inter-loop dependencies, given enough resources this entire loop could be executed at the same time.

We used R (the number of registers available) as a limit on the number of copies of the loop we could unroll. Since only one additional register is necessary for each copy of the unrolled loop we can unroll the loop R times.

By assuming that the serial machine will have "balanced capabilities" we expect that if there are enough registers for such unrolling, there are respectively enough resources to execute these loops all at once (ILP).

After examining the behavior with several different values of R, we took for specifics $R = 128$ in our comparisons. This R value was explicitly used in the loop unrolling analysis for the serial code and implicitly used in the parallel code analysis where we claim that the machine is assumed to be balanced (i.e., sufficient functional units and registers). The improvement in the serial execution for larger R values is very small.

4.2. Parallel Randomized Coin-Tossing Algorithm (No-Cut)

Two issues arise when analyzing the randomized coin-tossing algorithm: what is the behavior of the forward iterations (such as their convergence) and finding a bound for the length of the longest chain of short-cut elements (before encountering another selected element).

4.2.1. Convergence of Iterations In the first stage of the algorithm *forward iterations* are performed, where a selection step is applied between the iterations, and a short-cutting (also referred to as pointer-jumping) process is applied starting at the selected elements, until another selected element is encountered.

The selection step selects a subset of the elements of the list, such that the expected number of the elements selected is one quarter and no two consecutive elements are selected. The selection of elements which are more than two apart is probabilistically independent.

The expected number of skipped elements in each iteration is therefore $\frac{3}{4}$.

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) then we make the simplifying assumption that the number of active elements in each iteration will be: $M_0 = N, M_1 \leq \frac{1}{4}N, \cdots, M_i \leq (\frac{1}{4})^i N$

The sum of this series, necessary for the summing of the work in the algorithm will be: $\Sigma M_i \leq 1.34N$.

The number of iterations necessary for the algorithm to finish (i.e., that the list size be 1 at the last iteration) satisfies $(\frac{1}{4})^{\#iter} N = 1$; this implies $\#iter = log_4 N = 0.5 log_2 N$.

To support this analysis we simulated the iterations of the algorithm for various sizes of inputs (N values). We repeated the simulation for each input size (N) 20 times, and looked at $E(\Sigma M_i)$ and $E(\#iter)$ that resulted. According to the simulations: $\Sigma M_i \leq 1.33N$ and $\#iter \leq 0.46 log_2 N$; both fully supporting our analysis.

4.2.2. 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 length of the longest consecutive number of elements that will be short-cut due to our coin-tossing selection mechanism (henceforth called "longest chain").

This bound is necessary in the performance analysis of the parallel No-Cut list ranking algorithm, 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:

- A **"chain"** (or sublist) 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).
- A **chain's "length"** is the number of elements *after* the initial "1" (i.e., the elements which will actually be short-cut).
- Given an element, $P(length \geq i)$ is the probability that the length of a chain starting at it is i, or longer.

Next we summarize the approaches used towards our goal:

- We prove the following claim. *Claim 1*: $P(length \geq logN + loglogN) \leq \frac{1}{N}$.
  This can verbally be described as: the probability for chains whose length is longer than $logN + loglogN$ is upper bounded by $\frac{1}{N}$.
- Finding an i such that $P(length \geq i) \leq \frac{1}{N}$
  and numerically (experimentally) solving the inequality with i which is linear in $log_2 N$, i.e., $i = C_1 * log_2 N + C_2$ (see Figure 3). This was done to gain intuition into the distribution of the random variable of the length of the longest chain.
- Simulating coin-tossing of N elements and experimentally finding the mean value of the longest chain.

We proceed to proving *Claim 1. The probability of the length of a chain* will be determined as follows:

- A chain will be of length $\geq l$ if $\forall i : 1 \leq i < l, next^i(x) is\ not\ selected$ where: (1) x is the selected element that starts the chain, and is considered to be the 0 (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".
- Consider a specific chain which starts at some element x. Observe that every chain is of length at least 1, since by definition the zero element (x itself) is selected; $next^1(x)$ is always "0" (according to our selection scheme) and is therefore never selected.
- $\frac{1}{4}$ 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 $next^4(x) = 0$. The situation is depicted in Figure 2.
- So the series of selected elements, after x, will be:

$$selected: \quad 0, \frac{1}{4}, \frac{2}{8}, \frac{3}{16}, \frac{4}{32}, \frac{5}{64}, \ldots, \frac{i-1}{2^i}, \ldots \tag{1}$$

$$i: \quad 1, 2, 3, 4, 5, 6, \ldots, i, \ldots \tag{2}$$

  See also 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.

- So the probability that a thread is of length $\geq i$ is the ratio of threads which are still active at element i out of all the elements:

$$i: \quad 0, 1, 2, 3, 4, 5, 6, \ldots, i, \ldots \tag{3}$$

$$P(length \geq i): \quad \frac{1}{4}, \frac{2}{8}, \frac{3}{16}, \frac{4}{32}, \frac{5}{64}, \frac{6}{128}, \frac{7}{256}, \ldots, \frac{i+1}{2^{i+2}}, \ldots \tag{4}$$

To be proved: $P(i) = \frac{i+1}{2^{i+2}} \leq \frac{1}{n}$

We assume $i = logn + loglogn$ then $P(i = logn + loglogn) = \frac{logn+lln+1}{4*2^{logn+lln}} = \frac{logn+lln+1}{4nlogn} = \frac{1}{4n} + \frac{lln+1}{4nlogn}$ since $loglogn \leq logn$ we have $P(i = logn + loglogn) \leq \frac{1}{4n} + \frac{1}{4n} + \frac{1}{4nlogn} = \frac{1}{n}(\frac{1}{2} + \frac{1}{4logn})$.

We need to prove that $\frac{1}{4logn} \leq \frac{1}{2}$ for the n values we are interested in, which is like proving that $logn \geq \frac{1}{2}$ for the n values we are interested in. This is true for every $n \geq \sqrt{2}$, and in particular for every $n \geq 2$.

To conclude, we proved *Claim 1*, that for $i = logn + loglogn$, $P(i) \leq \frac{1}{n}$ for every $n \geq 2$, which is the range of input sizes we are interested in.

Intuition into Distribution of Longest Chain

We try to find an i such that $P(length \geq i) \leq \frac{1}{n}$. Since $P(i) = \frac{i+1}{2^i+2}$, as shown above, we are trying to see how $P(i) = \frac{i+1}{2^i+2} \leq \frac{1}{n}$ behaves.

A few manipulations to this equation:

$$\frac{i+1}{2^{i+2}} \leq \frac{1}{n};\ log_2(\frac{i+1}{2^{i+2}}) \leq log_2(\frac{1}{n});\ log_2(i+1) - (i+2)log_2 2 \leq -log_2 n;\ log_2(i+1) - (i+1) \leq 1 - log_2 n$$

We rename our variable $i + 1 = x$, so our equation becomes: $log_2 x - x \leq 1 - log_2 n$; then $x = Clog_2 n$ for some $C > 1$ will solve the inequality for a large enough n. We will experimentally try finding such a C for the range of n that interest us in the next subsection, using simulations.

We then plotted this x as a function of $log_2 n$ as per our equation (i.e., numerical solution), and we can see that i is approximately linear in $log_2 n$, i.e., $i \sim C_1 * log_2 N + C_2$. This is presented in figure 3.

Simulations to Support the Longest Chain Bound

We ran (randomized) simulations for over 200 different input-size (N) values, each of which was repeated for 20 times. The input-size (N) values ranged from 10^2 to 10^7.

The coin-tossing was done using the *addrans* function from the *UNIX* mathematical library. The function provides uniform random additive numbers and is based on Knuth's discussion in [K81].

A brief description of the simulations:

```
Repeat for Various N-values
{ Some N-value is randomly chosen
  Repeat 20 times:
  { Toss a coin N times
    Find the longest chain that was formed [denoted Maxlen]
  }
  Calculate E(Maxlen)
  Calculate E(Maxlen)/log2(N) [denoted R]
}
Find MAX(R) over all the different N-values
```

The simulations resulted in the following:

- We ran over 200 different N-values in the range: $10^2 \leq N \leq 10^7$
- $Max(R) < 1.23$
- Only 6 simulations resulted in $R \geq 1.20$
- Practical conclusion: $E(Maxlen) = 1.2 * log_2 N$

4.3. The Cut-6 Algorithm

An issue that arises in analyzing the Cut-6 algorithm is the sequence which describes the convergence of the forward iterations.

4.3.1. Convergence of Iterations In the first stage of the algorithm *forward iterations* are performed, where a selection step is applied between the iterations, and a short-cutting (also referred to as pointer-jumping) process is applied starting at the selected elements, up to at most 6 elements ahead, unless another selected element is encountered earlier than that.

The selection step used selects a subset of the elements of the list, such that on the average one quarter ($\frac{1}{4}$) of the elements are selected and no two consecutive elements are selected. The selection of elements which are more than two apart is probabilistically independent.

The first element will always be skipped and therefore contributes $\frac{1}{4}$ of the total number of elements to the count of skipped elements. The second element will contribute $\frac{3}{16}$, the third will contribute $\frac{4}{32}$ and so on.

The number of skipped elements in each iteration is therefor $[\frac{1}{4}+\frac{3}{16}+\frac{4}{32}+\frac{5}{64}+\frac{6}{128}+\frac{7}{256}]=\frac{183}{256}$.

Let N be the initial input size, and let M_i be the number of active elements in the i-th iteration. Then, the number of active elements in each iteration will be: $M_0 = N, M_1 \leq \frac{73}{256}N, \cdots, M_i \leq (\frac{73}{256})^i N$

The sum of this series, necessary for the summing of the work in the algorithm will be:

$$\Sigma M_i \leq N + \frac{73}{256}N + \cdots + M_i \leq \frac{1}{(1-\frac{73}{256})}N = \frac{256}{183}N \leq 1.4N$$

The number of iterations necessary for the algorithm to finish (i.e., the list size is 1 in the last iteration) satisfies $(\frac{73}{256})^{\#iter}N = 1$; this implies $\#iter = log_{(256/73)}N = 0.552 log_2 N$.

4.4. Wyllie's Parallel Algorithm

Wyllie's algorithms is relatively simple. Most of its work and time analysis is clear from the code and the counts presented. The only place we resorted to some additional assumptions was when we wanted to calculate efficiently the value of $log_2 N$, which is the number of iterations required by the algorithm.

Efficiently calculating $log_2 N$**.** It seems that the most efficient way to calculate $log_2 N$ (if the goal is shorten the critical time path) is by spawning a thread for every bit in the integer N and finding which of the bits is the most significant bit by shifting right. We assumed 64 bits integers, which would definitely be enough for the N values we look at. The $log_2 N$ calculation takes constant work, thus the large constant we get in the work term[2].

5. Compilation Feasibility

The thesis [D98] explains why the translation from the high level code (C and extended-C) to the optimized assembly code, which has actually been done manually, is feasible with known compiler techniques.

6. The Instruction Set

The explicit parallel instruction set is based on a standard (serial) instruction set (we used the *MIPS* version in [HP94]) with some additions. It includes a set of instructions that implement these new ideas and was appropriate for the problems we studied.

In [VDBN98] we described the new ideas and their implementation in the instruction set. Since a key feature of an instruction set model is implementability we also outlined mapping into the hardware for some of the novel constructs there.

The specifications for the explicit parallel instruction set (also referred to as spawn-join instruction set) can be found in the full paper version of [VDBN98] (see `http://www.umiacs.umd.edu/~vishkin/XMT/`). We include in appendix B a table which summarizes the new instructions.

7. Execution Model & Execution Times

We are interested in comparing performance results for the serial algorithm in a serial 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 and counting the work 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.

7.1. Execution Assumptions

Our performance results were derived as follows.
(i) Given the code, we tried to place each instruction as early as possible without violating data and control dependences.

[2]It might be possible, in the future, to calculate $log_2 N$ as a primitive instruction (which would be the most efficient way), thus eliminating the need for the above discussion.

(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 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, the sensitivity of our speed-up results to replacing 5 by slightly bigger or smaller numbers was marginal.
(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 (presented 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].

7.2. 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 is based on the *scheduling lemma* for XMT proved in [VDBN98].)

8. 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[3]	$15N + 4(\frac{N}{R}) + 5(N mod R) + 22$	$9N + 4(\frac{N}{R}) + 4(N mod R) + 22$
No-Cut[4]	$58N + 4log_2 N + 2$	$4.2{log_2}^2 N + 18.5 log_2 N + 27$
Cut-6	$80.5N + 2.2log_2 N + 2$	$44.7 log_2 N + 27$
Wyllie[5]	$10N log_2 N + 12N + 2log_2 N + 268$	$15log_2 N + 30$
RM[6]	$32N + 10log_2 N + 9$	$9{log_2}^2 N + 27log_2 N + 6$

8.1. Speed-up Results

We compared the execution time for the 4 algorithms (serial, No-Cut, 6-cut 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:

- The serial algorithm, under our assumptions and execution counts, is competitive for $P < 7$.
- Each of the three parallel algorithms has an area (of P-values and N-values) for which it is the best (see table in section C.1).

  When looking at a specific P (along a vertical section) we see, as expected, that:

  - For the larger N-values, the work-emphasized No-Cut algorithm "wins"
  - For intermediate N-values, the time-emphasized Cut-6 wins
  - For the smaller N-values, Wyllie's algorithm wins

Plots of the areas in which each algorithm is dominant are shown in figure 6, for parallelism values up to 10,000, and figure 5, for parallelism values between 1 and 50. This, as well as detailed results, are reported in Appendix C. Note that the table in section C.1 adds data about RM's algorithm and shows that for higher end of the larger N-Values it wins.

9. Other Implementations of List-Ranking Algorithms

In [Re94] both simple list ranking and weighted list ranking (referred to as list-scan) are 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

[3] 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).

[4] 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 $2log_2 N$ with high probability.

[5] 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).

[6] 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, 6-cut, 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 9.1.

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-scan (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 9.1 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 Figure 9.

As part of the *SB-PRAM* and *Fork95* projects ([KS97]) the PAD library of PRAM algorithms and data structures was developed. Amongst the problems studied was list ranking (see [Tr96]) for which several algorithms were implemented using *Fork95*. *Fork95* is a c-based parallel programming language. Three work-optimal deterministic EREW algorithms were implemented. All the algorithms switch at some experimentally determined parameter to Wyllie's algorithm (pointer-jumping). The execution was measured on a simulator for the SB-PRAM. The results reported are as follows: 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 $n = 10742$. The conclusion of the author is that from a practical point of view neither of the algorithms are fully satisfactory, due to large constants, although some randomized algorithm might achieve better results.

9.1. Detailed Comparison to RM'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, see the discussion in section 10).

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.

There are several issues that should be noticed 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 prefix-sum instruction described in Appendix A) 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 expect 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 ($32N + 10log_2N + 9$) is smaller than the one for the No-Cut algorithm (see the results in section 8), but the time term for the No-Cut algorithm is smaller than the one for RM's algorithm ($9log_2N^2 + 27log_2N + 6$). 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-scan 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. The hurdle (in the Cray implementation) is the need to group several sublists for amortizing unequal distributions of length. RM's solution used larger input length to overcome 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 appendix C. The analysis and simulations are presented in appendix G. The *most interesting conclusion* in this analysis is that the expected length of the longest chain in RM's algorithm is $\simeq 2.75 * log_2N$, while for the No-Cut algorithm Section 4.2.2 derived the result $E(Maxlen) \simeq 1.2 * log_2N$. This highlights an interesting advantage of the No-Cut algorithm; this advantage is due to its different randomized mechanism for picking "selected" elements.

10. Improving the Results

The idea of accelerating cascades (see [CV86a]) is to start with one algorithm, which gives best performance for the initial input size (N) and parallelism available (P), and then, during later iterations switch to another algorithm (or algorithms) as we advance on the computation and the input size decreases. The switching points between the algorithms is likely to be determined experimentally, as they depend on the advance of the iterations, the constant factors of each algorithm and the overhead of deciding to switch and the switch itself. Notice that other implemented parallel list ranking algorithms (described in section 9) switched to Wyllie's algorithm, which is considered to be the most efficient parallel algorithm for smaller lists at some experimentally determined point. For more on cascading, see Appendix D.

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.

[CV86] Cole R. and Vishkin U. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Computation* 70, 1986, 32-53.

[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.

[D98] S. Dascal. Experiments with List Ranking Algorithms for Explicit Instruction Parallelism. M.Sc. Thesis. Department of Computer Science, Tel Aviv University, August 1998.

[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. *Theoretical Computer Science* 6(2), 1996, 297-322.

[KS97] Kessler C.W. and Seidl H. The Fork95 parallel programming language: design, implementation, application. *International Journal of Parallel Programming*, 25(1), 1997, 17-49.

[K81] Knuth D.E. *The Art of Computer Programming, Vol. 2 - Seminumerical Algorithms*, 2nd Edition. Addison-Wesley. 1981.

[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 Im Stadtwald, Saarbrucken, Germany, December 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/` .

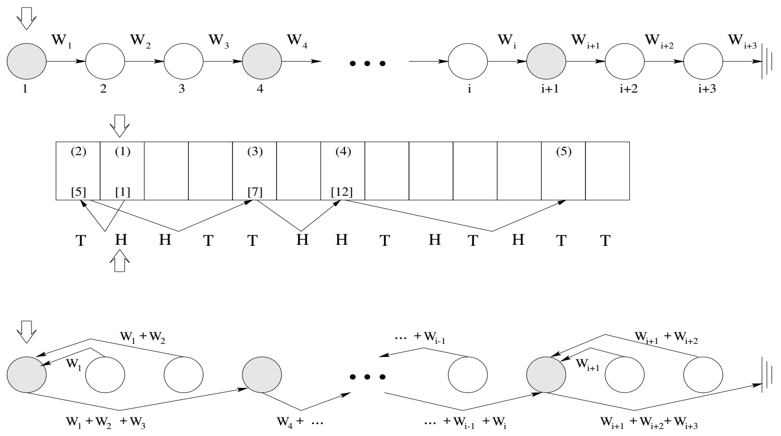

Figure 1: *Illustration of some common elements in the parallel randomized coin-tossing algorithms.* The upper figure depicts the original linked list, then comes an array representation of the linked list (the data-structure as kept in memory), and finally the linked list after one forward iteration of pointer-jumping. The down arrow $\Downarrow$ points at the head of the linked list (element number 1). w_i is the weight of the link between element i and element $i+1$. The elements are numbered according to their order in the linked list. In the array representation this numbering appears in round brackets; a pointer to the array index of the next element appears in square brackets. H stands for "head" and T stands for "tail". The shaded elements are the selected elements (Head followed by Tail). The bottom figure describes the outcome of a forward iteration. The backwards arrows include data that will later enable to compute the rank of the non-selected elements.

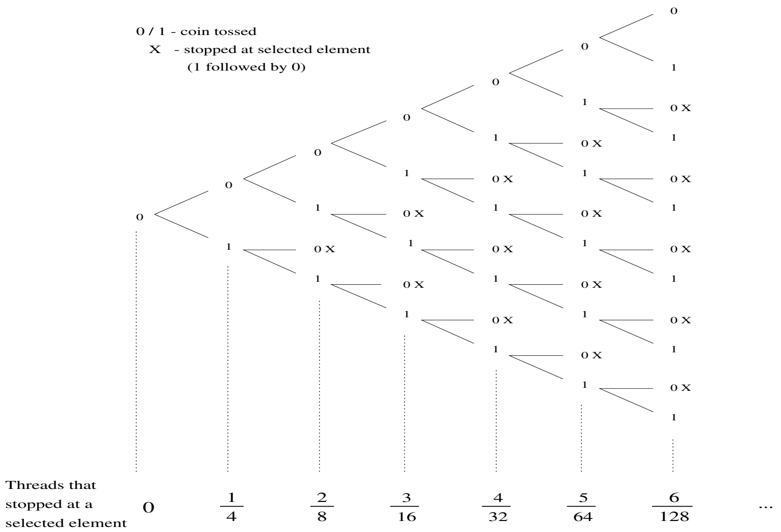

Figure 2: Probabilities of selected elements and of chain length

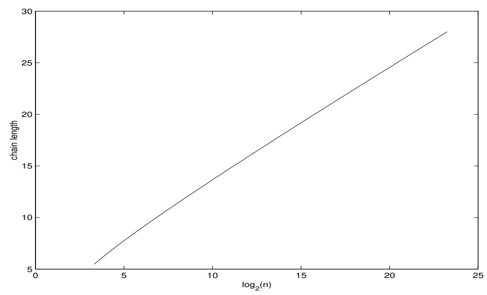

Figure 3: The chain length i for which $P(length \geq i) \leq \frac{1}{n}$ as a function of the logarithm of the input size (n) - numerical solution

A. Overview of the XMT programming model

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 Figure 4. 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, described below. The PS instruction is defined through grouping of "individual prefix-sum" instructions.

PS Ri Rj, an *individual prefix-sum* with respect to registers Ri and Rj, means the following inseparable ("atomic") outcome:

(i) add the value of register Ri to the value of register Rj and store the result in Ri, and

(ii) store the original value of Ri in Rj.

In and of itself, this instruction has an effect similar to a simple "add" instruction. However, the difference between the PS instruction and an "add" instruction is that several PS instructions may be cascaded into one *multiple-PS* instruction. For example, the sequence of k instructions:

$PS\ R1\ R2$; $PS\ R1\ R3$; ...; $PS\ R1\ R(k+1)$

performs the prefix-sum of the *base* $R1$ and the *elements* $R2, R3, ..., R(k+1)$ into $R2, R3, ..., R(k+1)$ and $R1$, respectively, which means the following:

$R2 = R1$; $R3 = R1 + R2$; ...;

$R(k+1) = R1 + R2 + R3 + ... + Rk$;

$R1 = R1 + R2 + R3 + ... + Rk + R(k+1)$.

In other words, the sequence of prefix-sum instructions becomes a multi-operand instruction.

The more interesting way (for this presentation) in which a multiple-PS instruction occurs in the assembly code is where each of the individual prefix-sum instructions (to be grouped into a multiple-PS instruction) comes from a different thread. Due to the assumed asynchrony among threads (i.e., IOS), any order of grouping individual prefix-sum instructions having the same base into a multiple-PS is acceptable. 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 a single-program multiple-data (SPMD). It is an extension of standard C which includes Spawn and multioperand instructions such as prefix-sum. Join instructions are not (explicitly) needed; following standard scoping in C, the scope of a Spawn command is enclosed between a pair of brackets; the closing bracket ending the scope of a Spawn command automatically implies a (hidden) Join. The Join comprises a parallel sum operation for monitoring the number of terminating threads.

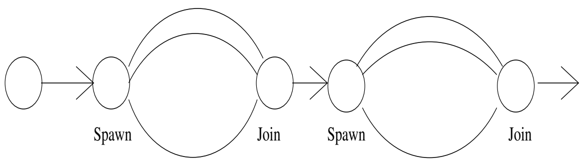

Figure 4: Parallel states from a Spawn to a Join and serial states

B. Explicit Multi-Threading (XMT): Instruction Set Quick Reference Table

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

Asynchronous mode

Instruction	*Instruction Meaning*
Spawn $Rb, j, Rn, REGS$	Switch from serial to parallel state. Spawn Rn threads, indexed $j, .. j + Rn - 1$. Execute by spawning the first (say) k threads the hardware allows. The hardware implementation generates the rest using Spawn-Recur. Assign $REGS$ physical registers to local virtual registers. Typically initialized to 0, global register Rb is a base register for the SUMI command of the matching Join.
Join Rb, Rn	Switch from parallel to serial state. Using a SUMI command, each thread increments global register Rb. Once $Rb = Rn$, switch to serial state.
$PS\ Ri, Rj$	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).
$SUMI\ Rb, Imm$	Given a sequence of such sum integer (SUMI) commands with the same R_b, the sum of the immediate values (Imm) is produced in parallel. $0 \leq Imm \leq 3$, Imm is inserted at compile time, and the sequence takes unit time for $\leq k$ instructions.
$PSM\ 15(Ri)Rj$	*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.
$lwa\ R3, 24(R1), 4[R2]$	Array addressing mode. Demonstrated by an array load instruction: $R3\ =\ Memory[R1 + 4R2 + 24]$.

C. Experimental Results

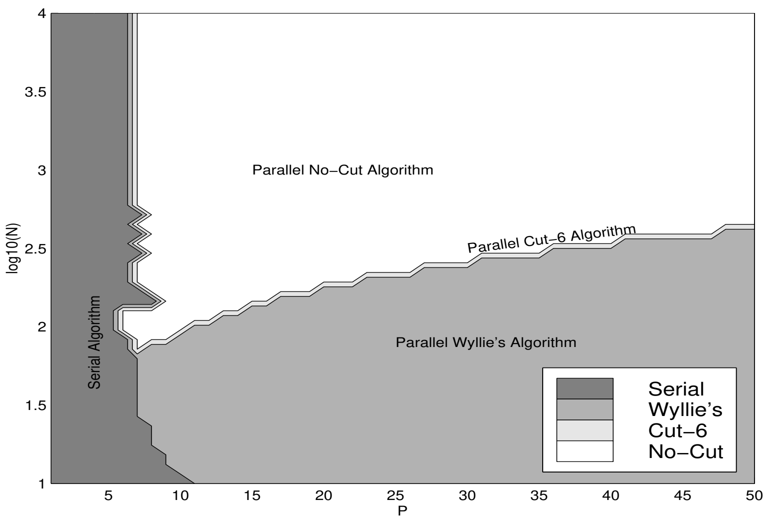

Figure 5: The areas in which each of the four algorithms for list-ranking is dominant. Parallelism between 1 and 50.

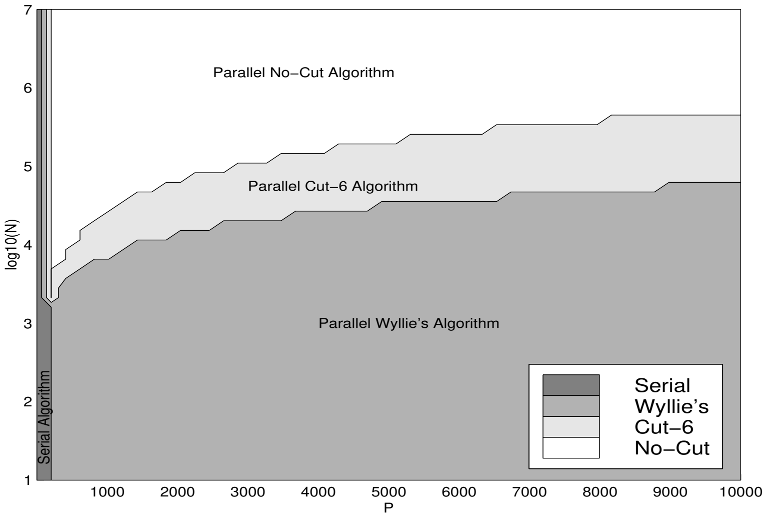

Figure 6: The areas in which each of the four algorithms for list-ranking is dominant. Parallelism up to 10,000.

C.1. Results Summary - Best Speedups Without RM's Algorithm

Below 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.

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

Algorithms Used:

- *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.

C.2. Results Summary - Best Speedups and Where RM's Algorithm Could Dominate

Below are the best speedups obtained when comparing the upper-bound on the parallel algorithms with the lower-bound of the serial algorithm. Recall that the RM results 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

Algorithms Used:

- *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;
- *alg 5* - reid-miller's algorithm.

C.3. Speedups Per Algorithm

Speedup for each of the algorithms, when comparing the upper-bound on the parallel algorithm with the lower-bound of the serial algorithm.

C.3.1. Coin-Tossing Algorithm (No-Cut) Speedup

N input size	P 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.44	2.11	2.50	2.75	2.93	3.06	3.25	3.37	3.64	3.82	3.88	3.91	3.92
10^3	1.47	2.68	3.69	4.55	5.29	5.93	6.99	7.83	10.31	12.72	13.79	14.19	14.40
$5*10^3$	1.51	2.94	4.28	5.55	6.76	7.90	10.02	11.93	19.33	30.79	38.36	41.79	43.74
10^4	1.53	3.01	4.44	5.83	7.17	8.46	10.94	13.26	23.09	41.57	56.70	64.52	69.30
$5*10^4$	1.55	3.09	4.61	6.12	7.62	9.10	12.02	14.89	28.53	63.30	106.60	138.09	162.02
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
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
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

C.3.2. Time Emphasized Algorithm (Cut-6) Speedup

N input size	P 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.17	1.82	2.24	2.52	2.73	2.89	3.12	3.27	3.64	3.90	3.99	4.02	4.04
10^3	1.11	2.10	3.00	3.81	4.55	5.22	6.40	7.41	10.82	14.95	17.12	18.00	18.47
$5*10^3$	1.11	2.18	3.23	4.25	5.24	6.21	8.06	9.83	17.46	32.73	46.19	53.53	58.15
10^4	1.11	2.21	3.29	4.36	5.41	6.44	8.46	10.43	19.46	40.52	63.39	78.09	88.32
$5*10^4$	1.12	2.24	3.35	4.46	5.56	6.66	8.85	11.03	21.67	51.50	95.14	132.60	165.09
10^5	1.12	2.24	3.36	4.47	5.58	6.69	8.91	11.11	22.02	53.54	102.42	147.21	188.40
10^6	1.12	2.24	3.36	4.49	5.61	6.73	8.97	11.21	22.39	55.78	110.93	165.46	219.38
10^7	1.12	2.24	3.37	4.49	5.61	6.73	8.97	11.22	22.43	56.06	112.04	167.95	223.79

C.3.3. Wylie's Algorithm Speedup

N input size	P 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.62	2.66	3.48	4.13	4.67	5.13	5.84	6.38	7.84	9.11	9.63	9.82	9.92
10^3	0.97	1.77	2.55	3.31	4.04	4.76	6.12	7.41	12.92	23.55	32.55	37.30	40.25
$5*10^3$	0.78	1.44	2.10	2.76	3.41	4.05	5.34	6.60	12.71	29.01	50.94	68.15	82.02
10^4	0.73	1.35	1.97	2.58	3.20	3.81	5.03	6.24	12.19	29.00	53.95	75.71	94.85
$5*10^4$	0.63	1.16	1.70	2.24	2.77	3.31	4.38	5.45	10.77	26.55	52.21	77.10	101.26
10^5	0.59	1.10	1.61	2.11	2.62	3.12	4.14	5.15	10.20	25.25	50.02	74.41	98.43
10^6	0.50	0.93	1.35	1.78	2.21	2.64	3.49	4.34	8.62	21.42	42.74	64.03	85.28
10^7	0.43	0.80	1.17	1.54	1.91	2.28	3.02	3.75	7.45	18.53	36.99	55.45	73.90

C.3.4. RM's Algorithm Speedup

N input size	P 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.46	1.78	1.92	1.99	2.04	2.08	2.12	2.15	2.21	2.25	2.26	2.27	2.27
10^3	2.16	3.41	4.23	4.80	5.23	5.56	6.03	6.36	7.12	7.68	7.88	7.96	7.99
$5*10^3$	2.55	4.66	6.43	7.93	9.23	10.36	12.22	13.71	18.10	22.42	24.35	25.07	25.45
10^4	2.66	5.03	7.16	9.08	10.82	12.40	15.18	17.54	25.43	34.85	39.75	41.71	42.76
$5*10^4$	2.78	5.47	8.08	10.60	13.05	15.43	19.98	24.27	42.55	77.64	107.07	122.56	132.12
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	199.07
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
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

D. Cascading the Algorithms

Since we have four different algorithms each of which is better in a different range of N and P values, it would be interesting to cascade several of them to get better execution times. We could, for example, start with the best algorithm for the initial N and P values (which we can know using a simple calculation of the upper bound), and then at each following iteration check to see which algorithm would do best for that N_i and P_i values.

In our optimizations chain, if we look at the performance for a fixed parallelism (P), we expect the No-Cut algorithm to perform best for largest N values, followed by the Cut-6 algorithm, then by either Wyllie's algorithm or the serial algorithm. Notice that for very small lists, we expect the serial algorithm to achieve best performance since it is work optimal. At the end of the cascading we can choose to run either Wyllie's algorithm or the serial algorithm (but not one followed by the other), since both of them advance until they reach the final ranking.

We should remember that our algorithms behave as follows. Let W_a and T_a be the work and time for algorithm A (No-Cut), W_b and T_b be the work and time for algorithm B (Cut-6), and W_c and T_c be the work and time for algorithm C (Wyllie's). We have a situation where $W_a < W_b < W_c$ but $T_a > T_b > T_c$ (see table 8).

Since the No-Cut and Cut-6 algorithms each advances in iterations and since each iteration reduces the size of the list it is working on by cascading the algorithms as the list size decreases, we would gain better overall performance. The overall performance depends also on some additional issues that need to be addressed (and that adds to the work we need to invest to perform this cascading) such as: (i) the overhead of checking when to switch between the algorithms; (ii) the overhead of switching from one algorithm to another; (iii) the setup of the algorithm we are switching to. Note that the code for such an involved algorithm would be longer (instruction cache size considerations might be relevant).

We started working on this issue. However, while there is merit in performance optimization, the separate basic algorithms are clearer than a more complex variant. So, for simplicity of presentation we didn't pursue this involved issue in full.

Initial Results As a flavor of the benefit of cascading the algorithms, we present two plots. In figure 7 we present the values of the relative gain from switching from the No-Cut algorithm at a specific list-size (N value), which is calculated as follows:

$$\frac{(Alg_1(N,P) - Alg_2(n,P))}{Alg_2(N,P)}$$

where Alg_1 is No-Cut and Alg_2 is the best algorithm in that specific area (e.g., Cut-6, Wyllie's or the serial algorithm). Note that the values are based on the evaluation of each of the algorithms separately, and take into account a single switch which occurs at that specific list-size (N value). In cases where two switches are beneficial (for example, no-cut → Cut-6 → Wyllie's) a more complex calculation, which is dependent on the list-size in several iterations, is necessary. In figure 8 we present, in a similar manner, the values of the relative gain from switching from the Cut-6 algorithm.

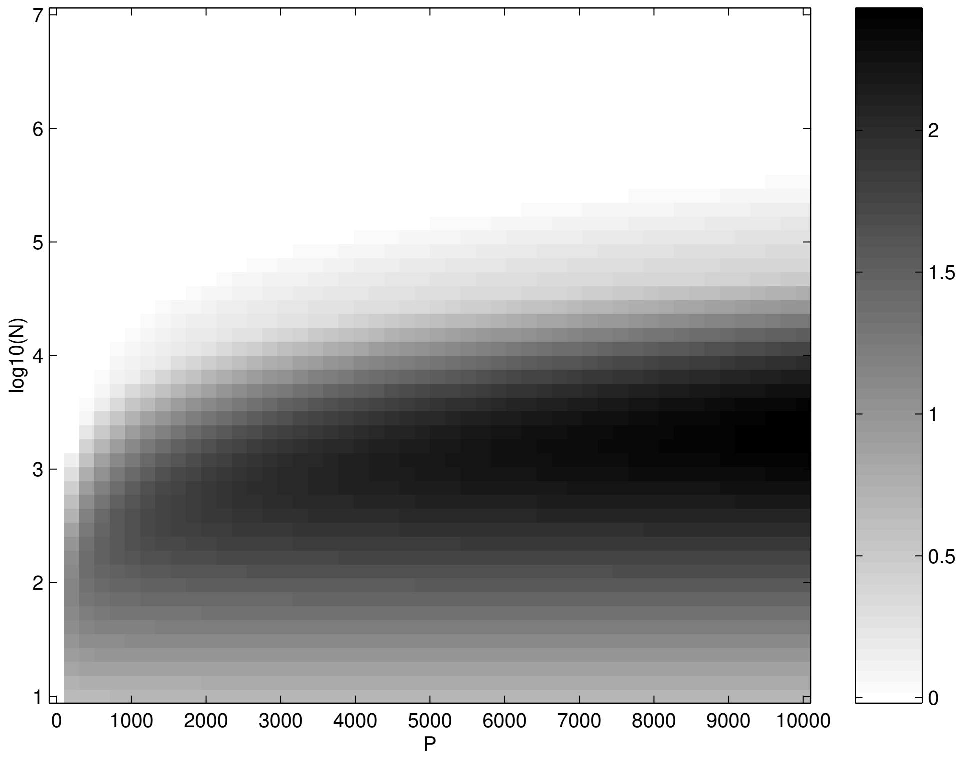

Figure 7: The relative gain from cascading from the No-Cut algorithm to Cut-6, Wyllie's or to the serial algorithm. The right column is a legend for the relative gain figures.

E. List Ranking Algorithms - Details

Additional details of the various algorithms is presented in pseudo-code.

E.1. Serial Algorithm

Details of the head finding, forward loop and final ranking in the serial algorithm follow.

Serial List Ranking

{
/* Find first element (HEAD) in the list: */
for $(0 \leq i < ListSize)$
if $(prev(i) == 0)$ /* then i is the HEAD */
{
$ListHead = i$
break
}

/* Forward Loop (finding HEAD's rank): */
$current = ListHead$
$next = next(ListHead)$
$max\_rank = 0$

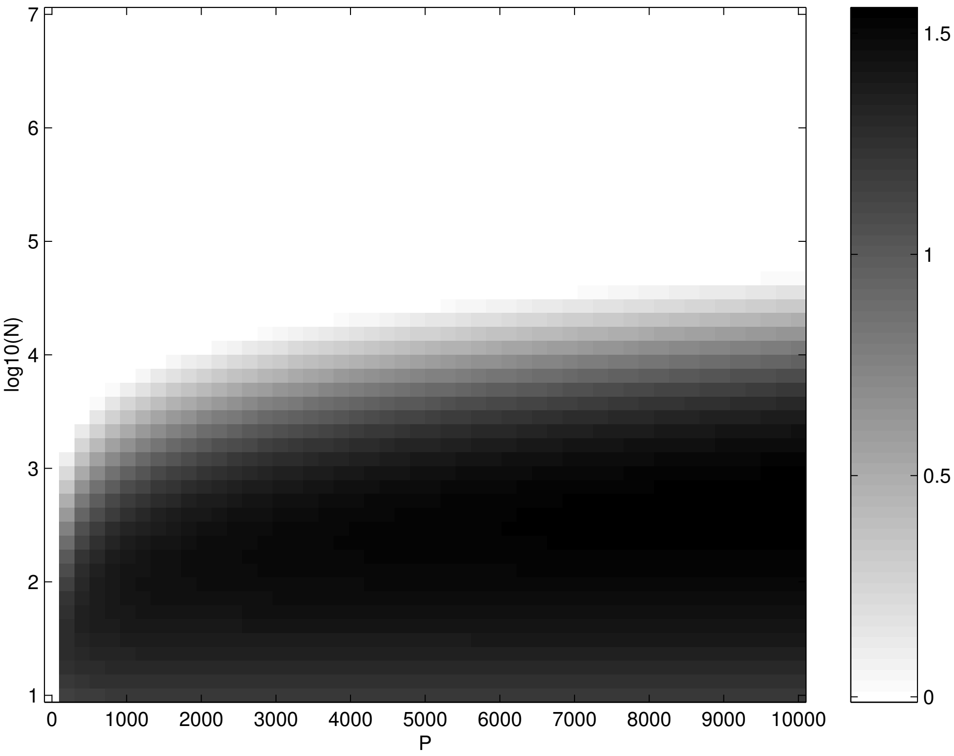

Figure 8: The relative gain from cascading from the Cut-6 algorithm to Wyllie's or to the serial algorithm.

while $(next \neq 0)$
{
$rank(current) = -1 * max\_rank$
$max\_rank = max\_rank + weight(current)$
$current = next$
$next = next(current)$
}

/* Final Ranking: */
for $(0 \leq i < ListSize)$
{
$rank(i) = rank(i) + max\_rank$
}
}

E.2. Parallel Randomized Coin-Tossing Algorithm (No-Cut)

Details of the selection step used, the forward iterations and the backward iterations used in the No-Cut algorithm follow. Note that in the forward iterations the random selection step is done separately for the first iteration, at the beginning before the while loop, and then at the end of each iteration for the *next* iteration. This enables us to utilize the synchronization point of the previous iteration (at the end of the spawn) for the random selection step of the next iteration, thus saving the need for an additional spawn if the selection was done at the beginning of each iteration.

Additional notations used in the pseudo-code

$\$$ - thread ID, given to each thread when spawned
selected - 1 if element selected, per element
parent - Pointer to the "ranking parent" element in the list, per element.
The final rank of a skipped element is determined relative to

the rank of its ranking-parent:

$rank(i) = rank(ranking\_parent) - ranking\_difference$

Parallel "No-Cut" List Ranking

```
{
  /* Forward iterations: */
  Random selection step based on coin-tossing; /* once for first
                    iteration, see later */
  while (curr_list_size > 1) /* condition to finish */
    {
      /* curr_actv is the array of active elements per iteration */
      SPAWN (0, curr_list_size) /* The spawn command creates
                    curr_list_size parallel threads whose IDs, denoted $, are each
                    an integer between 0 and curr_list_size - 1 */
        {
          if (curr_actv[$] is selected)
            {
              /* Note that variables next and rank are local to each thread */
              next = next(curr_actv[$])
              rank = rank(curr_actv[$])
              /* run until reach next selected element */
              while (next NOT selected)
                { /* skipping (short-cutting) non-selected elements */
                  rank = rank + (skipped_element's_rank)
                  update rank-difference, ranking-parent, etc. for skipped element;
                  shrink into "non-active array" using prefix-sum for indexing;
                  next = next(next)
                } /* finished short-cutting */

              /* SELECTED elements only - preparations for next iteration */
              random selection process based on coin-tossing;
              shrink into active array (using prefix-sum);
            }
        } /* end of SPAWN - this is a JOIN instruction terminating the threads
                    of the SPAWN, switching back to a single thread mode;
                    it is implicitly implied at the end of the spawn block */

      save information needed in the backward iterations (such as ranking parent);
      update information for next iteration, including curr_list_size;
    }

  /* Backward iterations: */
  while (nactv_size < 0) /* reconstructing original list */
    {
      /* curr_nactv is the array of the skipped elements per iteration*/
      SPAWN (0, nactv_size)
        {
          rank(curr_nactv[$]) = rank(ranking_parent(curr_nactv[$])) - rank(curr_nactv[$])
        }

      update information for next iteration (such as nactv_size);
    }
}
```

The Selection Step

```
{
  if (curr_actv[$] is the HEAD (first element) in the list)
     selected($) = TRUE
  else if (next(curr_actv[$] == NULL) /* LAST of the list */
     selected($) = FALSE
  else /* all the "middle" elements */
     {
        using an array of randomly tossed bits (provided)
        if (($ is 1) AND (($ + 1) is 0))
           selected($) = TRUE
     }
}
```

E.3. Time-Emphasized Algorithm (Cut-6)

The selection step used and the backward iterations are identical to those presented in the previous section. Details of the forward iterations follow.

Additional fields used in the algorithm per element

$ - thread ID, given to each thread when spawned
selected - 1 if element selected
done - 1 if element was "skipped" (i.e., pointer-jumped over) in this iteration
parent - Pointer to the ranking-parent element in the list.
The final rank of a skipped element is determined relative to the rank of its ranking-parent:

$$rank(i) = rank(ranking\_parent) - ranking\_difference$$

Parallel "Cut-6" List Ranking

```
{
  /* Forward iterations: */
  Random selection step based on coin-tossing; /* once for first
                          iteration, as in the No-Cut algorithm */
  while (curr_list_size > 1) /* condition to finish */
     {
        /* curr_actv is the array of active elements per iteration */
        SPAWN (0, curr_list_size)
           {
              if (curr_actv[$] is selected)
                 {
                    skipped = 0
                    next = next(curr_actv[$])
                    rank = rank(curr_actv[$])
                    /* run until reach next selected element OR at most skipped 6 elements */
                    while ((skipped < 6) AND (next NOT selected))
                       { /* skipping (short-cutting) non-selected elements */
                          rank = rank + (skipped_element's_rank)
                          update rank-difference, ranking-parent, etc. for skipped element
                          Set done = 1 for skipped element
                          next = next(next)
                          skipped = skipped + 1
                       } /* finished short-cutting */
                 }
           } /* end of SPAWN - synchronize */
```

```
SPAWN (0, curr_list_size)
  {
    if (curr_actv[$] is done) /* this element was skipped */
      {
        shrink into non-active array using prefix-sum for indexing;
      }
    else /* this element is still active */
      {
        /* preparations for next iteration */
        random selection process based on coin-tossing;
        shrink into active array (using prefix-sum);
      }
  } /* end of SPAWN - synchronize */

save some information for the backward iterations;
update information for next iteration, including curr_list_size;
}
}
```

E.4. Wyllie's Parallel Algorithm

- Instead of having two spawn-join blocks (i.e., an additional synchronization barrier) for each iteration of the algorithm we used alternating read and write arrays, so that every thread reads the information from the previous iteration, but writes the updated information into a separate buffer (which later becomes the input, read buffer, for the next iteration).This will reduce the dependence between the threads and allow us to utilize the independence of order property in the asynchronous mode to its fullest. At the end of the iteration the arrays are swapped.
- In order to be able to alternate between the read and write buffers without having to update pointers too frequently we use an indexed list (i.e., use the array-index of the successor as opposed to pointing to it in the buffer).
- Since our original (input) list consists of pointers to the successors we first transform the list into an indexed list.
- We need to calculate the number of iterations necessary: $log_2 N$. Only then can we perform the forward iterations.

Details of the algorithms iterations follow.

Wyllie's Parallel List Ranking

```
{
 /* Preparing an index-based array: */
 SPAWN (0, ListSize)
   {
     /* $ is the thread ID */
     if ((next($) ≠ 0) /* not LAST */
       { /* figure out the index for next */
         next_index = index(next($))
       }
     copy element to read_list($)
     next(read_list($)) = next_index /* use index instead of address! */
   }

 /* The pointer-jumping iterations: */
 for (1 ≤ k ≤ ⌈log2 ListSize⌉)
   {
```

```
SPAWN (0, ListSize)
  {
    /* read from read_list and write to write_list */
    if (next(next($)) ≠ 0) /* next is not LAST, from read_list */
      { /* update in write_list */
        rank($) = rank(read_list($)) + rank(read_list(next($)))
        next($) = next(read_list(next($)))
      }
  } /* end of SPAWN - synchronize */
swap the READ & WRITE lists for next iteration;
}
}
```

F. Optimization Considerations

F.1. Serial Algorithm

An alternative serial algorithm that was considered consisted of initially creating a doubly linked list (each element points to its next and previous elements), and then traversing the list from the end, while ranking the elements.

Both the alternative algorithm and the algorithm we selected have to traverse the list once and have to access each element another time (not in the serial "list" order). The main difference is that while the selected algorithm only needs the ranking difference value when accessing the elements out of order, and can then calculate their final rank, the alternative algorithm needs to follow pointers to reach its neighbors in the initial setup (double linking the list), which in our model results in more cache misses. The traversing of the list would be very similar in both cases.

F.2. Coin-Tossing Algorithm - Optimization for Smaller Work Count

Parallel algorithms usually produce more work than the serial algorithms for the same problem. Often, this extra work is needed to explicate parallelism and to coordinate parallelism during execution, shortening the critical time path of execution.

For a parallel algorithm to be competitive with the serial algorithm for practical input sizes and relatively small parallelism (P values) we need to reduce the amount of work in the parallel algorithm as much as possible, while still getting good parallel time ($T(n)$) values.

In the No-Cut algorithm for example, we copy the full contents of the elements to the current "active array" (thus accessing the data through a single address, saving the need to trace an additional pointer to the data) to minimize our penalties for cache misses within each thread, and to allow quick direct array access to the information in the elements.

The No-Cut algorithm is optimized for better work. It has better work constants (in the term $W(n)$) than the time-emphasized Cut-6 algorithm, but its asymptotic time bound includes a $O(log_2{}^2 N)$ term. Since the dominant work term is $O(N)$ the algorithm becomes dominant (in the term $\frac{W}{P} + T$) over the time term for relatively small inputs, and therefore its constant is very important.

Why does the "No-cut" algorithm perform less work? The issue at hand is comparing the work requirements of the no cut-algorithm, which short-cuts elements until the next selected element with algorithms that stop after short-cutting some constant number, say k, of elements (before reaching the next selected element). First note that the No-Cut algorithm gets some "chains" that are longer (implying threads with a longer critical time path). However, the scheduling lemma in [VDBN98] implies that we do not get penalized in terms of work for that in the XMT model, where each thread advances independently. The key observation is that there is a significant decrease in the number of iterations (first forward and then backward) necessary. We have longer threads but fewer of them in fewer iterations, so on balance the overall work term is smaller.

A single spawn-join block We considered having several shorter spawn-join blocks in the forward iterations as opposed to having a single longer block where everything (short-cutting, "copy-shrinking" in preparation for the next iteration and the selection step of the next iteration) is done. For smaller work and time counts it is better to have threads which are a little longer, thereby saving the overhead of re-loading local registers, etc. Since the copy-shrinking of "skipped" elements does not delay the short-cutting we can save potential cache-misses from re-loading by copying the skipped elements immediately after they have been short-cut. In addition, at the end of the short-cutting we already have the necessary information for the copy-shrinking of the active elements for the next iteration in local registers, and therefore can save additional cache-misses by doing it in the same thread. The selection step for the next iteration can be done at the same time that the copy-shrink of the active elements, thus saving us an additional, unnecessary synchronization barrier.

F.3. Coin-Tossing Algorithm - Optimization for Smaller Time Count

For larger input size (N values) we can utilize more parallelism (when higher levels of P are available) by reducing the work term $\frac{W(N)}{P}$. On the other hand, for smaller values of N the time term $(T(N))$ becomes dominant for smaller values of P. Therefore we are willing to have slightly more work done, and we utilize this additional work to shorten the critical path (i.e., improve the time optimization).

The parallel time-emphasized Cut-6 algorithm is optimized for better time (than the No-Cut coin-tossing algorithm). It has a smaller time term, $O(log_2 N)$, than the work-emphasized No-Cut algorithm, but has a larger constant for the work term.

Cut-k Algorithms That is, stop short-cutting after at most k elements. From this cut-k group of algorithms we get several spawn-join blocks and more iterations relative to the No-Cut algorithm, but each of the threads is shorter. We need more than one synchronization barrier since after short-cutting we need to find out which elements are still active (were selected and remain active, or were simply not short-cut due to stopping after k elements).

Since the number of elements we short-cut is at most k, the time term will now be $O(log_2 N)$. The additional work we have to invest isn't too large.

Choosing Cut-6 That is, among the family of cut-k algorithms we chose the one that stops after short-cutting at most 6 elements. As reported in Section 4.3, 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.

The constants for $k = 3, 4, 5, 6, ..., 11$ were respectively for the time term $49.5, 43.6, 41.4, \underline{40.7}, 41.9, 43.4, 45.4, 47.5, 49.8$ and for the work term $100, 92, 87.3, \underline{85}, 83.8, 83.4, 83, 82.6, 82.6$. The underlined values are for $k = 6$. Note that after these experiments were done we refined some of the execution assumptions, so the values presented here for comparison are not exactly the same as the final constants presented in the results.

Local copy of "next element" We tried implementing further time optimizations by copying the entire contents of the next-element in the current element's active array entry. The rationale being that we would have the immediate access to the next_element in the same memory block, thus saving on time consuming cache-misses. Unfortunately the improvement in the critical time path was very small, while the additional work necessary to do that was very large (in the $O(N)$ term), thus making this attempt non-competitive for the ranges we are interested in.

F.4. Wyllie's Algorithm

Wyllie's algorithm has a short critical time path while its work term $O(N log_2 N)$ is very large. It is therefore dominant for smaller input size and large P values, beating the Cut-6 algorithm. We tried optimizing it where each thread can advance independently (usually it is presented in the PRAM model, with its lock-stop synchrony).

Read and Write arrays Instead of having two spawn-join blocks (i.e., an additional synchronization barrier) for each iteration of the algorithm we used alternating read and write arrays, so that every thread reads the information from the previous iteration, but writes the updated information into a separate buffer (which later becomes the input, read buffer, for the next iteration). This will reduce the dependence between the threads and allow us to utilize the independence of order property to its fullest. At the end of the iteration the arrays are swapped. Since we already have the necessary information (such as the new rank value) in local registers (and we don't have to write it to a temporary place) this saves us both time and work of the re-initializing and re-loading necessary if we used two separate spawn-join blocks (for calculating all the updated ranks and then for updating them in the same copy of the list).

Using indices instead of pointers At the beginning of the algorithm we have some necessary setup, which is done once, in order to have an indexed list (as opposed to a pointer list). This is necessary so we can alternate between the read and write buffers (see previous paragraph) without having to unnecessarily update pointers all the time.

G. RM's Randomized Algorithm - Analysis and Simulations

In this section we present a "paper and pencil" analysis of the behaviour of RM's algorithm (see [Re94]) picking $k = \frac{n}{4}$ elements randomly at each iteration as heads of the sub-lists. Following that, is a summary of the results we obtained from simulations of the algorithm's behaviour using the same k.

G.1. Paper & Pencil Analysis

Two issues arise when analyzing RM's randomized algorithm: what is the behaviour of the forward iterations (such as their convergence) and finding a bound for the length of the longest chain (i.e., sub-list) of short-cut elements (before encountering another selected element).

First we take a look at the probabilities of an element to be selected and not selected in RM's selection scheme when picking $k = \frac{n}{4}$ elements randomly as heads of sub-lists. Then we will derive from that some reasonable length for the longest chain of consecutive elements to be short-cut. Later on we will derive, by looking at the convergence of iterations, the number of iterations the algorithm should run on an input list of size n, and the sum of the active elements in all the iterations (necessary to analyze the amount of work necessary).

We wish to find the probability that an element is selected , i.e., picked at least once during the $k = \frac{n}{4}$ draws.

We randomly pick a number x_1 from 1 to n with uniform distribution. So the probability that a specific element was picked in this one draw is $\frac{1}{n}$. This means that the probability that a specific element was *not* picked in this one draw is $1 - \frac{1}{n}$.

We repeat this picking step $k = \frac{n}{4}$ independent times. So the probability that a specific element is *not* selected at all during all the draws, denoted as p_{ns}, is:

$$(1 - \frac{1}{n})^{\frac{n}{4}} = ((1 - \frac{1}{n})^n)^{\frac{1}{4}} \underset{n \to \infty}{\longrightarrow} (\frac{1}{e})^{\frac{1}{4}} = 0.779$$

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 (i.e., picked at least once during the $k = \frac{n}{4}$ draws), denoted as p_s, is:

$$(1 - 0.779) = 0.221$$

G.1.1. Finding a Bound for the Length of the Longest Chain In order to analyze the execution time for the algorithm we need to find a realistic bound on the length of the longest consecutive number of elements that will be short-cut due to the selection mechanism used (also denoted as "longest chain" for short).

This bound is necessary in the performance analysis of the algorithm, 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.

Our goal below is to show that the length of the longest chain is around $log_2(N)$ with very high probability. Some definition follow:

- A **"chain"** (or sub-list) is a series of consecutive (pointer wise) elements in a list starting at a selected element and ending just before the next selected element.
- A **chain's "length"** is the number of elements *after* the selected element (i.e., the elements which will actually be short-cut).
- Given an element, $P(length \geq i)$ is the probability that the length of a chain starting at it is i, or longer.

The probability of length of a chain will be determined as follows:

- A chain will be of length $\geq l$ if

$$\forall i : 1 \leq i < l, next^i(x) is\ not\ selected$$

  where:

1. x is the selected element that starts the chain, and is considered to be the 0 (zero) element of the chain.
2. $next^i(x)$ is the i-th successor of element x.

- We already showed earlier that the probability of an element to be selected is $p_s = 0.221$ and the probability of an element to be *not* selected is $p_{ns} = 0.779$.
- So the probability that a chain is of length $\geq i$ will be:

$$p_s * (p_{ns})^i$$

meaning one selected element followed by i not-selected elements (at least).

We assume that the longest practical chain (i.e., sub-list) is the chain for which $P(length \geq i) \leq \frac{1}{n}$, and we proceed to finding such an i (i.e., its length).

An i for which $P(length \geq i) \leq \frac{1}{n}$ can be found using a few manipulations of this inequality. Since $P(length \geq i) = p_s * (p_{ns})^i$, as shown above, we are trying to see how $P(length \geq i) = p_s * (p_{ns})^i \leq \frac{1}{n}$ behaves:

$$p_s * (p_{ns})^i \leq \frac{1}{n}$$

$$(p_{ns})^i \leq \frac{1}{p_s} * \frac{1}{n}$$

$$log_2((p_{ns})^i) \leq log_2(\frac{1}{p_s} * \frac{1}{n})$$

$$i * log_2 p_{ns} \leq log_2 \frac{1}{p_s} + log_2 \frac{1}{n} = -log_2 p_s - log_2 n$$

Since $p_{ns} < 1$ then $log_2 p_{ns} < 0$ so we get:

$$i \geq \frac{log_2 n + log_2 p_s}{-log_2 p_{ns}}$$

$p_s < 1$ too, so we can ignore it since we're looking for a bound, and $-log_2(p_{ns}) = -log_2(0.779) = 0.360$ we have:

$$i \geq \frac{log_2 n}{0.360} \simeq 2.77 * log_2 n$$

We will experimentally back this assumption for the range of n that interest us in the next sub-section, using simulations.

G.1.2. Convergence of Iterations In the first stage of the algorithm *forward iterations* are performed, where a selection step is applied between the iterations, and a short-cutting process is applied starting at the selected elements, until another selected element is encountered.

The selection step selects a subset of the elements of the list, such that the expected number of the elements selected is 0.221 (as shown earlier). The selection of elements is probabilistically independent.

The expected number of skipped (i.e., short-cut) elements in each iteration is 0.779 (as shown earlier).

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) then we make the simplifying assumption that the number of active elements in each iteration will be:

$$M_0 = N, M_1 \leq 0.221N, \cdots, M_i \leq (0.221)^i N$$

The sum of this series, necessary for the summing of the work in the algorithm will be:

$$\Sigma M_i \leq N + 0.221N + \cdots + M_i \leq (0.221)^i N + \cdots \leq \frac{1}{(1 - 0.221)} N = \frac{1}{0.779} N \leq 1.29N$$

The number of iterations necessary for the algorithm to finish, i.e., that the list size be 1 in the last iteration (of course cascading is in order, but it is separated for comparison purposes), is:

$$(0.221)^{\#iter} N = 1$$

$$\Longrightarrow \#iter = log_{\frac{1}{0.221}} N \simeq 0.46 log_2 N$$

To support this analysis we simulated the iterations of the algorithm for various sizes of inputs (N values). We repeated the simulation for each input size (N) 20 times, and looked at $E(\Sigma M_i)$ and $E(\#iter)$ that resulted. According to the simulations:

$$\Sigma M_i \leq 1.28N$$

and

$$\#iter \leq 0.43 log_2 N$$

both fully supporting our analysis.

G.2. Simulations Results

We ran (randomized) simulations for over 200 different input-size (N) values, each of which was repeated for 20 times. The input-size (N) values ranged from 10^2 to 10^7.

The random picking of numbers between 1 and N was done using the *addrans* function from the *UNIX* mathematical library. The function provides uniform random additive numbers and is based on Knuth's discussion in [K81].

A brief description of the simulations:

```
Repeat for Various N-values
{ Some N-value is randomly chosen
  Repeat 20 times:
  { Pick a value between 1 and N with uniform probability N/4 times
    Remove the values that appear more than once (uniqueness)
    Find the longest chain that was formed [denoted Maxlen]
  }
  Calculate E(Maxlen)
  Calculate E(Maxlen)/log2(N) [denoted R]
}
Find MAX(R) over all the different N-values
```

The simulations resulted in the following:

- We ran over 200 different N-values in the range: $10^2 \leq N \leq 10^7$
- $Max(R) \leq 2.75$
- Less than 5% of the simulations resulted in $R \geq 2.70$
- Practical conclusion:
  $E(Maxlen) \simeq 2.75 * log_2 N$

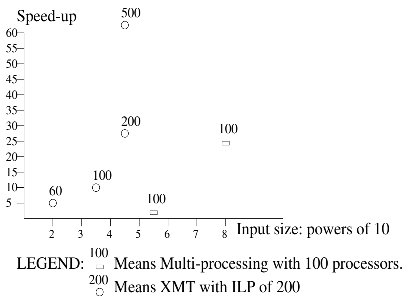

Figure 9: Comparison to [Si97b].