Final Course Paper on *Parallel Sorting of Intransitive Total Ordered Sets*

Johannes Singler

December 16, 2003

Abstract

For the problem of sorting an Intransitive Total Ordered Set, a combination of two parallel algorithms already known by literature is proposed and implemented. The benchmarks show significant speedup on a shared-memory multi-processor system.

1 Introduction

Sorting of *transitive total ordered sets* is one of the best investigated topics in Computer Science. A generalization of this is sorting an *intransitive total ordered set* which is more complex. While a transitive total ordered set (e. g. a subset of the real numbers) holds $a \geq b \wedge b \geq a \Rightarrow a \geq c$, this does not have to be true for a intransitive total ordered set. Only $a \rightarrow b \dot{\vee} b \rightarrow a$ is claimed. Nevertheless it can be proved that there is always an ordering that satisfies $a_1 \rightarrow a_2 \rightarrow a_3 \ldots \rightarrow a_n$. A good example for such a set is a tournament (e. g. in sports). Each competitor either wins or loses against every other competitor. But even if it is known that Player A defeats Player B who defeats Player C in turn, nothing can be said safely about the result of the match Player A has against Player C. Actually, the relation in such a intransitive ordered set is often referred to as a *tournament*.

The symbol for the relation be $\prec$. The tournament can be represented as a directed graph where every player corresponds to a vertex. Then $a \succ b$ is represented by an edge from vertex a to vertex b. If so, there must not exist a directed edge from b to a since this would violate the assumptions for an intransitive total ordered set. When represented as a graph, sorting a tournament is equivalent to finding a Hamiltonian path in this graph. Given a graph, a Hamiltonian path is a path through which the graph visits each vertex precisely once. I has been proven by Redei [4] that there is always such a path in every tournament. Unlike the regular sorting problem, there might exist more than one appropriate solution.

2 Literature Review

2.1 Core Papers

In [11] Wu proposes an algorithm for sorting an intransitive total ordered set with cost $\Theta(n \log n)$. The run time of this algorithm is $\Theta(n)$ with $\Theta(\log n)$ processors under the EREW PRAM model. Wu uses a new data structure called semi-heap for his algorithm. The definition of as semi-heap is as follows, using a new definition of maximum elements

out of three players/vertices A, B and C based on $\succ$. If one of the players beats both the others, he represents the only maximum entity. If every one of the three players wins once and loses once, all elements are maximal. This is formally stated by $a = max_{\succ}\{a, b, c\}$ if both $max\{a, b, c\} = b$ and $max\{a, b, c\} = c$ are false.

A semi-heap is then a complete binary tree similar to a regular heap except the different heap property. For every node n and its left and right children l and r respectively holds $n = max_{\succ}\{n, l, r\}$. Using a semi-heap, a sorting algorithm is suggested that always takes out the maximum element of the semi-heap and rebalances it correspondingly to satisfy the semi-heap property. Unlike the regular heap sort, it is not possible to replace the root of the tree by the least element since it may happen that the least element beats all the top three elements. When pushed to top it can neither be taken out in the next step nor pushed down the tree. Therefore, the former root element must be replaced by one of its children and the tree must be reordered top-down afterwards. This takes only $\Theta(log\ n)$ time because the algorithm only descends one branch and is thus bounded by the tree height. Hence sorting the tournament sequentially takes $\Theta(n\ log\ n)$ time.

Furthermore Wu proposes how to parallelize the algorithm mentioned above. The first step is buidling the semi-heap which has the same complexity as building a regular heap, namely $\Theta(n)$. So there is no parallelization needed for this step. Afterwards, one processor is assigned to each level of the binary tree. The replace operation is now pipelined. This is feasible since changes are always done top to bottom. So rebalancing the tree can be done "lazy". Every processor alternates between an *active step* and a *passive step*. At an active step a processor first performs local updates, i. e. reordering the nodes to satisfy the semi-heap property. After that it sends messages to the two adjacent processors above and below it (if they exist) to tell them about the changes. This is necessary because to avoid concurrent access the algorithm must keep two copies of each node in memory for the two processors that may have to change that particular node. At a passive step, each processor receives messages from its neighbors for use in the next active step.

In [10] Wu and Olariu present another algorithm concerning intransitive total ordered sets which is closely related to the problem mentioned above. They propose a cost-optimal solution for *merging* two intransitive ordered sequences. Traditional methods like bitonic merging or merging using ranking [2] can not be applied as they rely on the transitivity property.

The proposed algorithm uses a divide-and-conquer approach. To merge the two sequences p and q, p is split in the middle at element $p[split]$. If this element beats the leftmost element of q, the right part of p must be merged with q in the next step and concatenated to the left part of p. When the rightmost element of q beats $p[split]$, the algorithm proceeds accordingly in a symmetric way. If none of the two former possibilities applies, the algorithm performs a binary search to find a cutting point such that $q[cut]$ beats $p[split]$ and $p[split]$ beats $q[cut + 1]$. Since the role of p and q are alternated in the subsequent recursive merging process, both the lengths of p and q are at least halved after every two consecutive rounds. If both the lengths of p and q fall below $log^2 n$, no futher splitting is performed but a traditional sequential merging algorithm is applied. So the number of leafs of the resulting recursion tree is bounded by $O(n/log^2 n)$. Binary search takes $O(n)$ time, hence merging two intransitive ordered sequences to one intransitive ordered sequence can be done in $O(log^2 n)$ on $O(n/log^2 n)$ processors using the EREW PRAM model.

However, the cost-optimal merge in the strong sense is still an open problem, i. e. it must be proved that no algorithm can use lesser cost than a certain cost-optimal one (e. g. the cost-optimal described earlier).

The article [1] is much older than the two papers by Wu et al. . Bar-Noy and Noar there present a general method to transform traditional algorithms for sorting by comparison to algorithms which compute a Hamiltonian path in a tournament which is equivalent to finding an intransitive total ordered sequence. This is done using minimal edge feedback sets. A minimal edge feedback set F in a directed graph $G = (V, D)$ is as minimal set (with respect to containment) of edges such that $G = (V, D - F)$ is acyclic. Thus, a minimal edge feedback set is the subset of the graph's edges that "contradicts" to it being transitive. When these edges are swapped (i. e. their direction is reversed), the order can be considered transitive and a traditional sorting algorithm can be used to compute the Hamiltonian path, i. e. to sort the sequence. As shown by the authors, there is a one-to-one correspondence between the set of minimal edge feedback sets and the set of Hamiltonian paths in any arbitrary tournament. It is stated that if there is an algorithm for finding an Hamiltonian path in a transitive tournament there also exists another algorithm that computes a Hamiltionian path in an *intransitive* tournament *with the same complexity.* This theorem holds for merging paths in an intransitive tournament, as well.

The only problem remaining is to find a minimal edge feedback set. A very complicated method to accomplish this in constant time is proposed by the authors. However, the disadvantage of this method is that it needs concurrent memory access. Therefor it will not run on an EREW PRAM.

The method described above is applied to Cole's mergesort [2] and Valiant's merging algorithm [9], yielding to algorithms that compute a Hamiltonian path in $O(n \log n)$ and merge two paths of length n in $O(\log \log n)$ time respectively both using $O(n)$ processors on a CRCW machine.

Soroker's paper [8] is a important paper for my project since it gives another fast algorithm for finding a Hamiltonian path in a tournament. It also includes an algorithm for a generalization of this problem, namely finding a Hamiltonian *cycle* in a tournament. Soroker states and proves that there is a Hamiltonian cycle in every *strong connected* tournament. This is in contrast to the decision whether an arbitrary graph is Hamiltonian being NP-complete.

The algorithm for finding a Hamiltonian path in this paper uses an interesting fact about tournaments: There always exists a "mediocre" player who has both won and lost many matches, each at least $\lfloor \frac{n}{4} \rfloor$ to be exact. A divide-and-conquer style algorithms is constructed on top of this. After having found a mediocre player, P, Hamiltonian paths H_1 and H_2 for the subgraphs $W(P))$ (the players who defeated P) and $W(P))$ (the players who were defeated by P) respectively are computed recursively in parallel. The merge step is trivially done by concatenating (H_1, P, H_2). Since finding the mediocre player has complexity $O(\log n)$ and there are $O(\log n)$ recursion steps, the overall time complexity of the resulting algorithm is $O(\log^2 n)$ when implemented on an EREW PRAM using $O(n^2/\log n)$ processors. Soroker states that this looks quite efficient since the size of the input is $\Theta(n^2)$, but as seen above this is not quite true. One does not even have to consider the largest part of the input.

The problem of finding a Hamiltonian *cycle* in a tournament is generalized to finding a Hamiltonian path using a specified endpoint which is called a *restricted Hamiltonian path* by the author. Furthermore, Soroker proves that a Hamiltonian cycle can be found by searching for a restricted Hamiltonian path starting at a sink, i. e. from a vertex from which all other vertices are reachable. The ends of this Hamiltonian paths are then connected together adequately to form a Hamiltonian cycle. An important computation required for this algorithm is splitting the tournament graph into strongly connected subgraphs.

Although the time complexity for accomplishing this task for an arbitrary graph is $O(n^{2.81})$ it can be done much faster for a tournament. The *score* of a vertex is the number fo vertices it dominates. First the algorithms sorts the *score sequence*, i. e. the list of all scores, in non-decreasing order $s_1 \leq s_2 \leq \ldots \leq s_n$. Then it calculates the partial sums $p_k = \sum_{i=1}^{k} s_i - \binom{k}{2}$ for $1 \leq k \leq n$. Finally the graph can be partitioned into strongly connected components just by looking for zeros in the sequence $p_1, p_2, \ldots, p_k$ and splitting it up accordingly.

2.1.1 Comparison and Conclusion

The algorithm proposed by Soroker in 1988 has total cost of $O(n^2 \; log \; n)$ compared to $O(n \; log \; n)$ for the cost-optimal method suggested by Wu in 2000 wherey both use the EREW PRAM model. Still the algorithm by Soroker is faster but it uses much more processors, namely $O(n^2/log \; n)$ compared to $O(log \; n)$. This is not good for practical use. The possibility of transforming any sorting algorithm to an algorithm for finding intransitive sorted sequences is of theoretic interest but has the drawback of needing concurrent memory access. Therefore it is not useful either in practice.

2.2 Auxiliary Literature

Reid and Beineke [5] give a review on the mathematical foundations of tournaments and present various former results about them. Some proofs are even presented in multiple versions. Lots of formulas deal with the numbers of possible (with respect to isomorphism) different tournaments for certain conditions. Some theorems concerning scores and the score sequence in a tournament can be found here, as well.

Hell and Rosenfeld consider further generalization of the problem of finding an intransitive ordered sequence in their paper [3]. The results of matches between consecutive players in the list has to match a prescribed pattern that states whether an edge must go "forward" or "backward". When all edges ought to go forward, i. e. v_i beats v_{i+1} for all valid i, this resembles to our main problem. Depending on the pattern, the complexity for finding a suitable path varies between $O(n)$ for an alternating pattern to $O(n \; log \; n)$ for the canonical pattern in the form $O(n \; log^{\alpha} \; n)$ where α varies between 0 and 1.

Another problem concerning tournaments is described in the paper [6]. A *king* is a vertex in the tournament graph from which every other vertex is reachable by a path of length at most 2. The presented algorithm looks for a sorted sequence as do the algoritms of Wu and Soroker, but even with the following side condition: Every vertex in the sequence must not only dominate its right neighbor, but must also be a king of the following sequence. For this reason, this is a specialization of sorting an intransitive ordered sequence and has complexity of $O\left(n^{3/2}\right)$.

3 (Former) Project Proposal

We would like to implement at least one of the presented algorithms for sorting an intransitive total ordered set. Since the algorithm by Wu seems to be the best one, this obtains priority while the algorithm by Soroker would take second place. A solution by applying the transformation proposed by Bar-Noy and Noar might be of theoretical interest but seems very hard to implement since it relies on the very complex algorithm of Cole.

In conclusion it would be very interesting to compare the first two of the algorithms above in terms of speed and cost.

4 Project

4.1 Evaluation of the Real-Life Applicability of the Described Algorithms

As stated above, the input to the problem of sorting an Intransitive Total Ordered Set is one bit for each pair of players in the tournament and two nodes in the graph respectively. So if this intransitive relation is arbitrary, it consumes $O(n^2)$ space. In contrast to this, Wu's algorithm for performing the sorting has the cost of only $O(n\ log\ n)$. There is also no redundancy in the input data because there is no transitivity. When excluding the symmetry of the matrix, not a single entry in the relation's matrix can be computed out of other entries. One may question how this is possible. In fact, Wu's algorithm does not consider all the input data. However, one cannot determine beforehand what "part" of the data might be needed. If the input data is actually stored in a matrix, any algorithm that generated this data already has complexity $O(n^2)$, so it would not be of much use to come up with a better algorithm for sorting the tournament after generating the data. However, the outcome of a game between two players could be inherent in their attributes, even if this is not as simple as a comparison and not transitive as well. The only condition that the relation must fulfill is anti-symmetry since, the reflexive edges are not actually looked at. The following is an example.

Let us assign a number to each player. The relation R_t where a player beats all the other players with numbers greater than his, would be transitive. But it can easily be made intransitive by calculating the exclusive or between the result of R_t and a boolean function s. s can be arbitrary, but it must be symmetric and must yield a boolean value for each pair (a, b) of players. This value should be computed only out of the attributes (in this case the numbers) of the players. So there is no further need for $O(n^2)$ memory to store the relation. This makes it possible to work on very large problems.

But still, the maximum problem size would be of magnitude 2^{30}. Since Wu's algorithm only uses $\lceil log_2(n) \rceil$ processors, a maximum of 30 processors could be used. Furthermore, only every other processor is used in a pipeline step. Therefore, at the most 15 processors can be occupied a 100% at one time with realistic problems. In addition to this, they would have to communicate with each other at every other pipeline step, which lasts only a few clocks. This would lead to much overhead and very bad cache foot print, since the data has to be moved between the processors all the time.

On the other hand, Soroker's algorithm utilizes a divide-and-conquer approach. Since the merge step is trivial, the data can be processed independently after the division step. Consequently, the problem can be divided up for arbitrary many processors quickly. But compared to Wu's algorithm, the cost of Soroker's solution is $O(n^2 log(n))$ which is pretty bad. So probably, the latter would run even more slowly on multiple processors compared to the former run on only one processor.

4.2 Proposal of a Combined Algorihtm

In conclusion, we propose a combination of the two algorithms introduced here. First, Soroker's algorithm is applied to split up the data into as many parts depending on the number of processors available. Then, on each processor, the cost-optimal agorithm invented

by Wu is processed to finish the sorting. As a result, this combination gets the best out of both possibilities.

We did not consider Bay-Noar's proposal to transform Cole's algorithm to the intransitive problem because it is said to be extremely hard to implement and to gain only poor real-life performance.

4.3 Implementation Details

In contrast to traditional (transitive) sorting algorithms, for constructing a random problem, it is feasible to generate a random relation. The initial sequence of players is not important at all. For this reason, we use just the ascending sequence of player numbers as the initial sequence.

To match realistic preconditions, we assume that the number of processors, p, is much smaller than n.

4.3.1 Dividing the Data by Using Soroker's Algorithm

For an overview, a step of Soroker's algorithm divides a sequence S into two parts. Situated in between them is the pivot element, which loses to all the players on its left side in S and wins over all the players in S on its right. But we must avoid linear degradation of the division process by coincidentally always choosing a player that loses or wins all its matches. Consequently, the algorithm looks for a player who has both won and lost at least $\lfloor \frac{m}{4} \rfloor$ games, where m is the number of players in the current part of the sequence to divide up, $|S|$. This search is performed in parallel by all available processors, where each processor inspects $\left\lceil \frac{m}{p} \right\rceil$ players. When the first processor that finds a feasible player, returns, all the other computations are also interrupted. The pivot element is now known.

Next, each processor is assigned to a part of S of equal size. Each processor then compares every element in its part to the pivot element and adds it either to its very own loser sequence or winner sequence. Additional memory of complexity $\mathrm{O}(m)$ in total is allocated before, to buffer these intermediate results.

After all the processors have finished this task, one of them can calculate the new positions of both the pivot element and all the loser and winner sequences, so that they do not overlap each other. Finally, all the processors are used to rearrange their data again from the temporary memory to the result sequence.

The process of dividing the sequence into two parts is continued recursively. Likewise, the set of processors to be occupied is divided up. So the computations become completely independent of each other.

4.3.2 Sequential Sorting Using Wu's Algorithm

When there is only one processor left for a certain part of the data set, it starts to sort it by using the sequential version of Wu's algorithm as described above. A semi-heap must be constructed which occupies again $\mathrm{O}(n)$ space. Although heap sort, for example, can be done in-place, this is not possible with the semi-heap, since it is no longer guaranteed to be a complete binary tree after the first step.

After every part has been sorted, all processors are synchronized using a barrier synchronization. There is no need for a merge step because all the processors have been working on the sequence which is returned as the result.

As a by-product of this project, a graphical demonstration program for Wu's algorihtm has been developed. It can be downloaded from the project website [7].

4.3.3 Pseudo-Code of the Combined Algorithm

The following pseudo-code denotes the combined algorithm. T is the set of players in the tournament, p the number of processors.

procedure SequentialSort(T) {Wu's algorithm.}
Build up Semi-Heap H from T.
Construct empty sequence S.
while H not empty **do**
  Add root element to sequence S.
  Replace root element recursively top-down.
end while
end procedure

procedure ParallelSort(T, p) {Soroker's algorithm.}
if p = 1 **then**
  return SequentialSort(T)
else
  Try to find a mediocre player v among the first h ones in T who has both won and lost at least $\lfloor \frac{n}{2.2} \rfloor$ games.
  if no player found so far **then**
    Find a mediocre player v who has both won and lost at least $\lfloor \frac{n}{4} \rfloor$ games.
  end if
  in parallel with p processors Compare all players in T to v. Count number of wins and losses.
  Construct table of where to move the players for the resulting sequence.
  in parallel with p processors Rearrange all the players so that v divides them into loosers L(T) and winners W(T).
  in parallel with p processors ParallelSort(L(T), p/2), ParallelSort(H(T), p/2)
  Merge (W(T), v, L(T)).
  COMMENTTrivial step.
end if
end procedure

4.3.4 Further Optimization and Fine-Tuning

The theoretical result by Soroker [8] only guarantees the existence of a player who has both won and lost at least $\lfloor \frac{m}{4} \rfloor$ games. So, in the worst case, the relation between the lengths of the two parts is only 1:3, which is not satisfying. Therefore, it might be advantageous to not stop the search after the first player found, but to look further for one that divides the sequence more evenly. The actual implementation looks for a player that both won and lost at least $\lfloor \frac{m}{b} \rfloor$ games whereby $2 < b < 4$. Then for $b = 2.2$, for example, one part is at most 22% longer than the other. However, the search is stopped after inspecting a certain number h of players and not having found a player that fits this stronger condition. Then, if the algorithm has not already run into one, a player that has fulfills the weaker condition is found. The described optimization led to much better splitting of random data for $b = 2.2$

and $h = 100$, so this feature was used during benchmarking.

Another way to attack the problem of uneven division is the following: One can divide up the processors in the same relation the sequence is divided. Hence, in the extreme case, only one fourth of the processors are assigned to the first part, and tree fourth to the second one. However, we figured out that it does not improve the performance, in particular not in combination with the trick described above. Thus, this feature is no longer part of the implementation.

4.3.5 Platform

The algorithm was implemented in the C programming language using the *pthread* library for any POSIX compatible operating system. The source code can be found on the project website [7]. All the threads are allocated before the actual sorting takes place. Also, one can choose between busy waiting and the usage of condition variables supported by the operating system to synchronize the threads. So the amount of overhead can be minimized.

4.4 Benchmarks

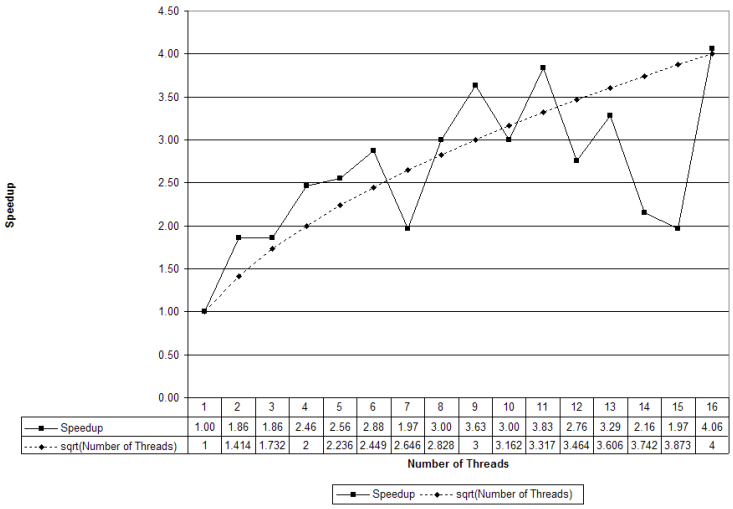
With the proposed algorithm, relative and absolute speedup are the same. When run with only one thread, there is no divide-and-conquer performed at all, the sorting is done sequentially. The effort to decide this consists of only one comparison of integeres and is therefore negligible.

4.4.1 Benchmark Configuration

The program was run on a SunFire 6800 with 20 processors and 20GB of memory. A tournament of 50000 players was sorted using 1 to 16 threads. The input data is generated before the actual sort step thus is identical for each run. To achieve more exact time measurement, for each number of threads, the sorting took place 10 times in a row. The wall clock time was divided by the number of runs afterwards.

4.4.2 Results

The achieved performance results are shown in figure [1]. Unfortunately, a linear speedup could not be achieved. The curve looks more similar to $O(\sqrt{p})$, which is also depicted for comparison in the figure. Also, the program performs best for the number of threads being a power of 2. This is not surprising since the set of players is divided into *two* parts in each division step. The reasons for performing the program so poorly are not obvious, but figure [2] gives a bit of clarification. Although all chunks of data have approximately the same size, the time needed to sort them differs heavily. Because the slowest thread counts when measuring the wall clock time, this of course hurts the performance. It happens even though the data is random and uniformly distributed. So apparently the performance of the sequential sorting algorithm using the semi-heap data structure is strongly data dependent. It cannot be anticipated by the algorithm which part of the data might be better or worse to sort for the follwing sequential algorithm. The reasons for the sequential performing so differently are unknown to us. Maybe it depends on the cache foot print of the data. There, slight differences might have heavy impact on the performance.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Speedup	1.00	1.86	1.86	2.46	2.56	2.88	1.97	3.00	3.63	3.00	3.83	2.76	3.29	2.16	1.97	4.06
sqrt(Number of Threads)	1	1.414	1.732	2	2.236	2.449	2.646	2.828	3	3.162	3.317	3.464	3.606	3.742	3.873	4

Figure 1: Benchmark Results

```
Sorting using semi-heap for 3086 elements took 0.030000 s.
Sorting using semi-heap for 3047 elements took 0.030000 s.
Sorting using semi-heap for 3074 elements took 0.030000 s.
Sorting using semi-heap for 3154 elements took 0.030000 s.
Sorting using semi-heap for 3159 elements took 0.030000 s.
Sorting using semi-heap for 3018 elements took 0.030000 s.
Sorting using semi-heap for 3070 elements took 0.040000 s.
Sorting using semi-heap for 3151 elements took 0.040000 s.
Sorting using semi-heap for 3209 elements took 0.040000 s.
Sorting using semi-heap for 3172 elements took 0.040000 s.
Sorting using semi-heap for 3116 elements took 0.040000 s.
Sorting using semi-heap for 3130 elements took 0.040000 s.
Sorting using semi-heap for 3120 elements took 0.040000 s.
Sorting using semi-heap for 3185 elements took 0.070000 s.
Sorting using semi-heap for 3146 elements took 0.070000 s.
Sorting using semi-heap for 3148 elements took 0.070000 s.
```

Figure 2: Timing Details.

However, the main reason is probably the overhead of Soroker's algorithm to divide the input data. It is actually designed to use $O(n^2)$ processors. So the speedup only increasing similar to $O(\sqrt{p})$ just fits when increasing the number of processors linearly.

5 Conclusion

Two parallel algorithms for the problem of sorting an Intransitive Total Ordered Set, known by literature, have been combined to achieve good real-life performance. A significant speedup could be achieved. However, it did not increase linearly with respect to the number of processors, but approximately proportional to $\sqrt{p}$. This is due to the utilized divide-and-conquer algorithm, which was actually designed to utilize $O(n^2)$ processors. At least, the speedup does not seem to be bound by a constant.

Further researchers might discover the bottle necks that harm the performance or try to use another method to divide up the player set. Then it would even be possible to use this combination of algorithms in the CGM model, i. e. without shared memory. However, if the input data is arbitrary, data of size $O\left(\left(\frac{n}{p}\right)^2\right)$ would have to be transferred to each

node after the division step.

References

[1] A. Bar-Noy and J. Naor. Sorting, minimal feedback sets and hamilton paths in tournaments. *SIAM Journal of Discrete Mathematics. 3, (1)*, pages 7–20, February 1990.

[2] Richard Cole. Parallel merge sort. *SIAM Journal on Computing, 4(1988), 770-785.*, 4:770–785, March 1988.

[3] Pavol Hell and Moshe Rosenfeld. The complexity of finding generalized paths in tournaments. *Journal of Algorithms*, 4:303–309, 1983.

[4] Redei. Ein kombinatorischer satz. *Acta Litt. Sci. Szeged*, 7:39–43, 1934.

[5] K. B. Reid and L. W. Beineke. Tournaments. In *Selected Topics in Graph Theory*, chapter 7, pages 169–204. Academic Press, New York, 1979.

[6] Shen, Sheng, and Wu. Searching for sorted sequences of kings in tournaments. *SIAM Journal on Computing*, 32(5):1201–1209, 2003.

[7] Johannes Singler. Project website for the prallel algorithms course project 2003. http://www.jsingler.de/COMP5704/.

[8] Danny Soroker. Fast parallel algorithms for finding Hamiltonian paths and cycles in a tournament. *Journal of Algorithms*, 9(2):276–286, June 1988.

[9] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, September 1975.

[10] Wu and Olariu. On cost-optimal merge of two intransitive sorted sequences. *IJFCS: International Journal of Foundations of Computer Science*, 14, 2003.

[11] J. Wu. On sorting an intransitive total ordered set using semi-heap. In *International Parallel and Distributed Processing Symposium*, pages 257–262, 2000.