ABSTRACT

Title of Dissertation: EMPIRICAL STUDIES IN PARALLEL SORTING

Evan Golub, Doctor of Philosophy, 1999

Dissertation Directed By: Professor William Gasarch,
Associate Professor Clyde Kruskal,
Department of Computer Science

I examine different parallel algorithms for sorting in rounds. Most of these algorithms use a graph to indicate the comparisons to be made. The primary difference between the algorithms is how these graphs are chosen. One uses graphs that are shown to exist using non-constructive techniques, several yield constructions of the required graphs, and one uses a randomized algorithm. The constructive algorithms would traditionally be preferred even though the processor requirements are higher. It is shown that the non-constructive algorithms can actually be used by generating the needed graphs using random number generators skewed appropriately.

EMPIRICAL STUDIES IN PARALLEL SORTING

by

Evan Golub

Dissertation submitted to the Faculty of the Graduate School of the
University of Mayland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1999

Advisory Committee:
Professor William Gasarch, Co-Advisor
Associate Professor Clyde Kruskal, Co-Advisor
Associate Professor Michael Franklin
Assistant Professor Jeffrey Hollingsworth
Professor Lawrence Washington

DEDICATION

I’d like to thank [in alphabetical order :)] all faculty, family, friends and students who inspired me, lived though and helped me live through this work...

On the research front, I’d like to thank David Arnow who ignited my first sparc of interest in parallelism and Bill Gasarch and Clyde Kruskal who re-ignited the candle.

TABLE OF CONTENTS

Introduction

Background

There are several interesting comparison based problems that are looked at in computer science: searching, selection, merging and sorting. These questions can be phrased as :

Searching :

> "If we are given a list of n ordered values, how many comparisons would be needed to determine whether a specific value was included in that list?"

Selection :

> "If we are given n values and a integer k , how many comparisons would be needed to find the k^{th} smallest value?"

Merging :

> "If we are given two ordered lists each having n values, how many comparisons would be needed to combine them into a single ordered list?"

Sorting :

> "If we are given n values in a list, how many comparisons would be needed to order those values?"

The ability to solve these questions quickly is an important one across many applications of computer science [Knuth1]. As the number of values being worked with grows, it is important to avoid having the run-time grow too quickly. This has been the motivation behind the desire to develop the most efficient possible algorithms for these problems. It should be noted that by “efficient” algorithms, we are only concerned with the number of comparisons done during the process.

These questions have been well studied in the sequential case (one processor working on the problem). For searching, the upper and lower bounds are known (trivially) to be n. For selection, the bounds do not yet match, but the upper and lower bounds have been brought quite close to each other: $2.97n$ [DZ1], [DZ2] and $(2+\frac{1}{2^{40}})n$ [DZ3]. For merging, the upper and lower bounds are (trivially) $2n-1$. For the comparison based sorting problem, the upper and lower bounds are $n\log n+\Theta(n)$ [see Knuth1].

These problems can be solved much faster when using parallel processing. When given multiple processors that can work together to solve these problems, two factors are often discussed: the number of processors used and the number of rounds used. In each round, each processor is able to execute a single comparison. Between rounds, the processors communicate their findings with one another. Within this domain, there are two standard optimization problems: (1) given a fixed number of values and processors minimize the number of rounds; (2) given a fixed number of values and rounds minimize the number of processors. While this model may not be realistic as it ignores the computation and communications between rounds, it is a well studied model. Since it is a well studied model, it serves as a good testing ground for the philosophies which I later present.

When solving these problems using multiple processors, a division can also be made between the situation where you have fewer processors p than you do values n ($p \le n$) and when you expect to have more processors than values ($p>n$). There is also the case in which you expect to have many more processors than values ($p>>n$) and therefore consider how many processors would be required to accomplish a job in a fixed number of rounds. These last two case are obviously related.

The following good upper and lower bounds are known for these problems in the first case:

p≤n

Searching	$\Theta\left(\frac{\log(n+1)}{\log(p+1)}\right)$ [Krus1]
Minimum	$\Theta\left(\frac{n}{p} + \log\log n\right)$ [Val1]
Selection	$O\left(\frac{n}{p} + \log n \log\log n\right)$ [Vish1][AKSS1], $\Omega\left(\frac{n}{p} + \log\log n\right)$ [Val1]
Merging	$\Theta\left(\frac{n}{p} + \log\log n\right)$ [Val1]

Sorting	$\Theta\left(\frac{n\log n}{p}\right)$ [AKS1][Cole1]

Figure I.1

When we move to the cases where p>n the searching problem becomes trivial (we can't use more than n processors to accomplish the task quicker than in a single round). Exact results are known to minimum finding and merging and can be inverted to determine the number of processors which would be required in a specific number of rounds.

p>n

Minimum	$\log\log n - \log\log\frac{p}{n} + \Theta(1)$ [Val1]
Merging	$\log\log n - \log\log\frac{p}{n} + \Theta(1)$ [Krus2]

Figure I.2

As an example of how one of these algorithms works in rounds, we can look at finding the minimum of a group of n values in two rounds using $O\left(n^{\frac{4}{3}}\right)$ processors. For the first round, we partition the original n values into $n^{\frac{2}{3}}$ groups of $n^{\frac{1}{3}}$ values.

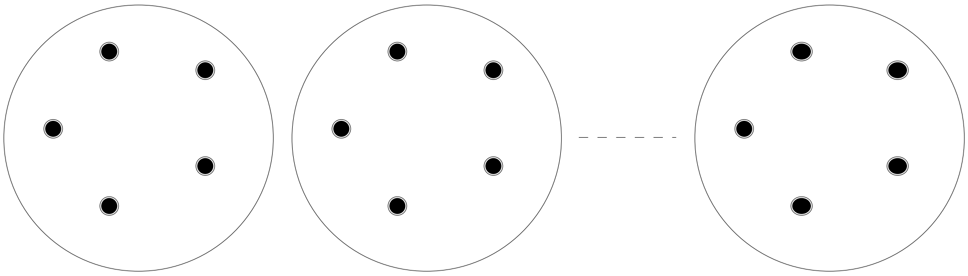

Figure I.3

In the first round, we compare all elements within a subgroup with all other elements in that subgroup. They can be represented as a complete graph connecting the values within a group. Each processor will be responsible for one of the comparisons represented by the edges of the graph.

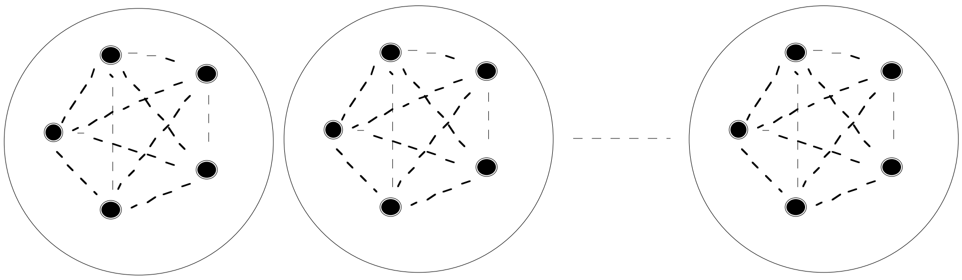

Figure I.4

This will require $O\left(n^{\frac{2}{3}}\right)$ processors per group, or $O\left(n^{\frac{4}{3}}\right)$ processors overall. After the first round, we will have enough information to find a minimum value for each group.

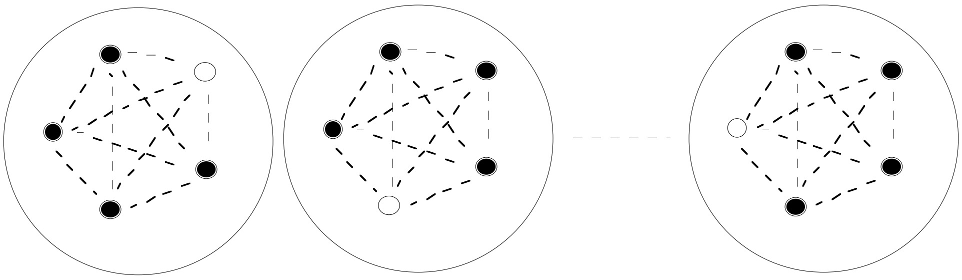

Figure I.5

We can then use the same $O\left(n^{\frac{4}{3}}\right)$ processors in the second round to compare all of these local minimums to one another in a single round to obtain the minimum value across the original *n* values.

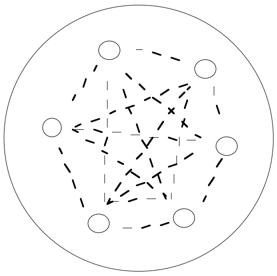

Figure I.6

If more than *k>2* rounds are made available, then the elements will be divided into groups based on the number of rounds. The first *k-1* rounds will be used to find the minimum in each of these groups using this same algorithm recursively. The final round will again be used to find the minimum of these minimums.

As an example of this, in the case of 3 rounds we would use an initial groups of size $n^{\frac{3}{7}}$. This would use the 2 round minimum finding algorithm on each of the groups, requiring $n^{\frac{4}{7}} \cdot O\left(\left(n^{\frac{3}{7}}\right)^{\frac{4}{3}}\right) \Rightarrow O\left(n^{\frac{8}{7}}\right)$ processors. We would then find the minimum of these minimums in the final round, requiring $\begin{pmatrix} n^{\frac{4}{7}} \\ 2 \end{pmatrix} \Rightarrow O\left(n^{\frac{8}{7}}\right)$ processors. We could similarly show that minimum finding in 4 rounds would require $O\left(n^{\frac{16}{15}}\right)$ processors. This generalizes as k rounds requiring $O\left(n^{2^k/2^k-1}\right)$ processors.

In the case where you have more processors available than values, the bounds are not as well defined for general selection or sorting.

The case of sorting is the one which will be the primary focus of this work. We will look at the question:

> "If we are given k rounds, and in each round each processor can ask one question, how many processors would be needed to sort n values?"

The answer to this question will be a function of the number of values n and the number of rounds k. This is the problem in which we are interested. With vast enough resources, we would like to be able to sort a group of values in very few rounds. Again, if an exact result were found for this, it could be inverted to answer the question of number of rounds for a fixed number of processors.

Previous results in area of interest

All of the algorithms which I will be discussing follow the same basic approach to sorting: within a given round elements are compared (one processor does one comparison) and between rounds information is extended using some form of transitive closure. In most of these algorithms, the comparisons to be executed during each round (except for the final round) are represented by a graph such that the vertices are the values and an edge existing between two vertices indicates that the two corresponding values will be compared during that round.

The results can also be divided into two basic categories; constructive and non-constructive. In the case of a constructive algorithm, an explicit set of instructions is given to construct the graphs (that in turn represent the comparisons that need to be done). In the case of a non-constructive algorithm, the existence of the required graph is proven (usually probabilistically) but instructions are not given for building the actual graph. In computer science, a constructive proof has traditionally been preferred since it can be used to create a program that will be able to solve the given problem. It has been felt that non-constructive proofs may not be easily translated into a program. I intend to challenge this viewpoint.

Another issue that merits attention is the work done between rounds in most of the above mentioned results. While each processor evaluates a single comparison within a round, the processors must communicate their results to one another and execute a transitive closure of these results to fully realize the information they have learned with those comparisons. While transitive closure does not involve any comparisons, in the implementation of these algorithms it represents a significant amount of computing time. Several papers discuss algorithms that do not require full transitive closure between rounds. While the number of processors required is slightly higher, the between-round processing time could be considerably less.

In the following summary of past results in this area Sort(n,k) denotes that we are ordering n values using k rounds of comparisons and full transitive closure between rounds while Sort(n,k,d) denotes that we are ordering n values using k rounds of comparisons and only d-level transitive closure between rounds.

HH1-81	$\mathrm{Sort}(n,k) = O\left(n^{\frac{3\cdot 2^{k-1}-1}{2^k-1}} \log n\right)$	• The algorithm is non-constructive. • This is the first algorithm in the area.

Figure I.7

HH2-82	$\mathrm{Merge}(n,k) = \Theta\left(n^{\frac{2^k}{2^k-1}}\right)$ $\mathrm{Sort}(n,k) = O\left(n^{1+\frac{2}{\sqrt{2k}}}\right)$	• The algorithm is constructive. • Better time asymptotically than HH81 results although the algebra is inexact. • $k \geq 3$ for sorting

Figure I.8

BT1-83	$\mathrm{Sort}(n,2) = O\left(n^{\frac{3}{2}} \log n\right)$	• The algorithm is non-constructive.
	$\mathrm{Sort}(n,2,d) = O\left(dn^{1+\frac{d}{2d-1}} (\log n)^{\frac{1}{2d-1}}\right)$	• Gives a 2 round algorithm that only requires *d*-steps worth of transitive closure. If you plug in d=2, you get $O\left(n^{\frac{5}{3}} (\log n)^{\frac{1}{3}}\right)$.
	$\mathrm{Sort}(n,2,d) = \Omega\left(n^{1+\frac{2}{2d-1}}\right)$	• If you plug in d=2, you get $\Omega\left(n^{\frac{5}{3}}\right)$.
	$\mathrm{Sort}(n,k,2) = O\left(n^{\frac{3}{2}+\frac{1}{2(2^m-1)}}\right)$ $k=2m-1$	• Gives an algorithm that only requires 2-step transitive closure (direct implication) for an odd number of rounds.

Figure I.9

Alon1-86	$\mathrm{Sort}(n,2) = O\left(n^{\frac{7}{4}}\right)$	• The algorithm is constructive. • It uses techniques in projective geometry to build graphs that can be used to sort fast. • It only requires direct implications rather than a full transitive closure.

Figure I.10

AAV1-86	$\mathrm{Sort}(n,k) = \Omega\left(n^{1+\frac{1}{k}}(\log n)^{\frac{1}{k}}\right)$ $\mathrm{Sort}(n,k) = O\left(n^{1+\frac{1}{k}}\right)$-expected value	• Gives a good lower bound for the problem. • Gives a constructive randomized algorithm.

Figure I.11

Pip1-87	$\mathrm{Sort}(n,k) = O\left(n^{1+\frac{1}{k}}(\log n)^{2-\frac{2}{k}}\right)$ non-constructive $\mathrm{Sort}(n,k) = O\left(n^{1+\frac{2}{k+1}}(\log n)^{2-\frac{4}{k+1}}\right)$ constructive	• This paper is a good reference due to its references to other works • The constructive upper bound uses Ramanujan graphs [LPS1]. • This was the first known subquadratic constructive solution (though it was *published* after Alon86).

Figure I.12

AA1-88	$\mathrm{Sort}(n,2) = \mathrm{O}\left(\dfrac{n^{\frac{3}{2}}\log n}{(\log\log n)^{\frac{1}{2}}}\right)$	• Non-constructive algorithm. • Gives a slight improvement over Pip87 but only does it for 2 rounds.

Figure I.13

Bol1-88	$\mathrm{Sort}(n,k) = \mathrm{O}\left(\dfrac{n^{1+\frac{1}{k}}(\log n)^{2-\frac{2}{k}}}{(\log\log n)^{\frac{k-1}{k}}}\right)$	• Gives only slight improvement over Pip87 but does it for general case rather than just 2 rounds as AA88 does.

Figure I.14

WZ1-93	$\mathrm{Sort}(n,k) = \mathrm{O}\left(n^{1+\frac{1}{k}+o(1)}\right)$	• Gives better constructive expanding graphs that can be used with Pippenger's algorithm.

Figure I.15

My contribution

I will address both the utility of non-constructive proofs and the impact of full versus partial transitive closure through empirical studies. I will discuss whether non-constructive proofs based upon the probability of the existence of a certain graph are any more difficult to code as computer programs than constructive algorithms.

There are two major philosophies that I wish to demonstrate:

- Non-constructive proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.
- Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.

Rather than using actual parallel machines to execute these experiments (since we do not currently have machines with the required number of processors to do so) I will simulate these machines on a single processor system. In the implementation of the algorithms, a comparison done in an iteration of a loop will be considered to be an iteration done on an individual processor. The information obtained in each loop will be stored and used in such a way that later iterations of a loop will not gain information from the earlier ones. This preserves the behavior of each iteration actually being done at the same instant on different processors.

I generate between 25 and 100 inputs for each algorithm for each configuration tested. In the case of probabilistic algorithms where graphs are generated at random, these 25 to 100 inputs will be a graph-data pair. After running the programs on these inputs, I take the maximum number of processors required over these runs for each configuration. I look at the individual runs for the first few configurations fed to each algorithm to determine whether there appears to be a need for more than the planned number of inputs.

In the following chapters I will address several of the algorithms that are mentioned in the earlier table. I will look at how the empirical results compare to the theoretical ones, discuss some of the limitations and even raise a few new questions.

Chapter 1 : Pippenger's K-Round Sorting Algorithm

Section 1.1: Sketch of algorithm

Pippenger [Pip1] presents non-constructive proofs of the following upper bounds :

$$\mathrm{Select}(n,k) = \mathrm{O}\left(n^{1+\frac{1}{2^k-1}}(\log n)^{2-\frac{2}{2^k-1}}\right)$$

$$\mathrm{Sort}(n,k) = \mathrm{O}\left(n^{1+\frac{1}{k}}(\log n)^{2-\frac{2}{k}}\right)$$

These algorithms use a-expanding graphs to represent the comparisons to be done in each round (except the final round that needs to compare all remaining unordered elements to each other). Pippenger shows two important things: (1) that after answering the questions represented by the a-expanding graphs, there are relatively few values whose position is not known, and (2) that relatively small a-expanding graphs exist. This work also has the advantage that it is shown that there is a high probability that a graph built using the techniques discussed will be an a-expanding graph.

Definition 1.1: *a*-expanding graphs

An *a*-expanding graph is defined as a graph in which for any two sets of vertices of size $a+1$, there is at least one edge between the two sets. ♦

Theorem 1.2[Pip1]: *a*-expanding graphs generate a great deal of information

Pippenger shows that if n elements are compared according to the edges of an *a*-expanding graph, then there will be at most $O(a \log n)$ candidates remaining for any given rank. ♦

Theorem 1.3[Pip1]: Small a-expanding graphs exist

Pippenger shows that small *a*-expanding graphs exist using a probabilistic argument. He shows that for large values of n, there is a high probability that a graph created by adding each possible edge with a probability of $p = \frac{2 \ln n}{a}$ will be an *a*-expanding graph. This translates to a high probability that *a*-expanding graphs with $O\left(\frac{n^2 \log n}{a}\right)$ edges exist. ♦

It should be noted at this point that the value of p is set without using order notation, but the results based upon it are. This implies that a constant factor might be used to fine tune the value of p in actual usage. This type of fine tuning will be discussed later.

Theorem 1.4[Pip1]: Selection in k rounds can be done with $O\left(n^{1+\frac{1}{2^k-1}}(\log n)^{2-\frac{2}{2^k-1}}\right)$ processors

With the existence of small a-expanding graphs proven, Pippenger goes on to show that for selection a good value for a is $\frac{n^{1-\frac{1}{2^k-1}}}{(\ln n)^{1-\frac{2}{2^k-1}}}$. This gives us a graph with $O\left(n^{1+\frac{1}{2^k-1}}(\log n)^{2-\frac{2}{2^k-1}}\right)$ edges. After comparing elements based on an a-expanding graph based on this value of a (and doing transitive closure based on the results) there will be $O\left(n^{1-\frac{1}{2^k-1}}(\log n)^{\frac{2}{2^k-1}}\right)$ candidates left in the selection problem. The values that have been eliminated have been determined to have either too many values above or below them for them to be in the position for which we are selecting. We do not know the order of the non-candidates relative to each other, just that they cannot be in the desired position. Since the selection problem takes two parameters (the values and the position you wish to select) it does not matter if the number of values that have been eliminated as candidates above and below is equal.

We can adjust the second parameter accordingly. By recursively doing selection based on the remaining candidates the result of select(n,k) = $O\left(n^{1+\frac{1}{2^k-1}}(\log n)^{2-\frac{2}{2^k-1}}\right)$ is obtained. ♦

Theorem 1.5[Pip1]: Sorting in k rounds can be done with $O\left(n^{1+\frac{1}{k}}(\log n)^{2-\frac{2}{k}}\right)$ processors

For sorting, Pippenger shows that a good value of a is $\frac{n^{1-\frac{1}{k}}}{(\ln n)^{1-\frac{2}{k}}}$. After comparing elements based on an a-expanding graph built with this value of a, the values can be partially ordered in such a way that there will be $O\left(\frac{n^{\frac{1}{k}}}{(\log n)^{\frac{2}{k}}}\right)$ groups, each having $O\left(n^{1-\frac{1}{k}}(\log n)^{\frac{2}{k}}\right)$ values, so that for any two groups a_i and a_j ($i<j$) all the values in a_i will be less than the values in a_j. Therefore, once each group is sorted, the entire set of values will be sorted. Again, by recursively sorting the remaining groups using the a-expanding graphs, the result of sort (n,k) = $O\left(n^{1+\frac{1}{k}}(\log n)^{2-\frac{2}{k}}\right)$ is obtained.♦

Section 1.2: Implementation of algorithms

Both the non-constructive selection and sorting algorithms are built using a-expanding graphs. The difference between the specific a-expanding graphs is the probability p that is used to place edges into the graph. I began by implementing the selection algorithm, then moved on to the sorting algorithm. Although my main interest is in sorting, the selection problem looked as though it would serve as a good testbed for my approach.

Section 1.2.1: Implementation of selection algorithm

I first implemented selection (specifically median finding) in two rounds, using the first round to compare the values using the a-expanding graph (based on $a = \frac{n^{1-\frac{1}{2^k-1}}}{(\ln n)^{1-\frac{2}{2^k-1}}}$), and then using the second round to compare all of the remaining candidates to one another.

To implement the algorithm, I needed the probability p which was to be used to generate the a-expanding graphs. This was based on the value for a that was in turn based on n. As a starting point, I took p to be the exact value specified in the paper. After gathering results on 2-round median finding for values of n ranging between 100 and 5000 (using that value), I charted the number of processors used in each of the two rounds in the worst case (over approximately 100 different inputs per size of n) and found that more work was being done in the first round than in the second. This implied that smaller graphs might be better. I generated results based on p multiplied by a constant factor between 0.1 and 2.0. These results led me to conclude that a good value would be somewhere between 0.2 and 0.4. I generated results more finely between 0.2 and 0.4 to find this value.

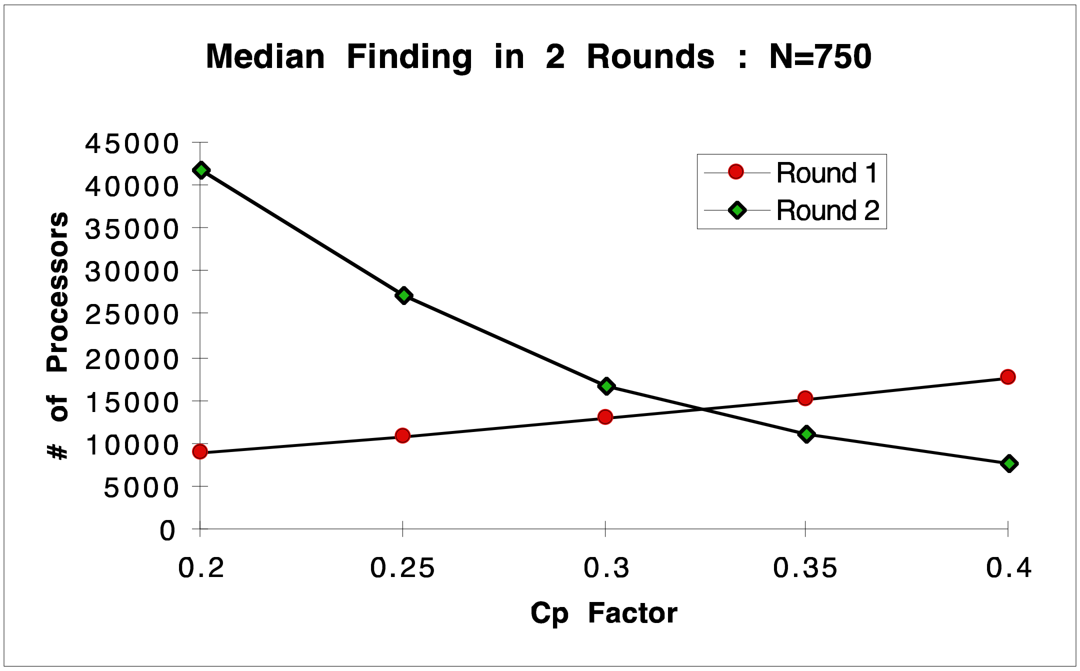

Figure 1.5

Figure 1.5 shows that when the graphs were generated with each edge being placed with a probability of $.32p$, the first and second rounds used roughly the same number of processors. Experiments showed that the number of processors used was consistent with the formula. Since my real interest was in the sorting results, I did not pursue testing for k round selection.

Section 1.2.2: Implementation of sorting algorithm

For sorting, the value for a is $\frac{n^{1-\frac{1}{k}}}{(\ln n)^{1-\frac{2}{k}}}$. Since the value for p is based on the criteria for an a-expanding graph rather than its use (sorting -vs- selection), that value remains $\frac{2\ln n}{a}$. In 2 round sorting, the first round of comparisons (based on the a-expanding graph) would be used to partially rank the values, and the second round would be used to fully sort each sub-group. The fact that the number of groups were "order of" rather than exact values posed an interesting question of how to identify the sub-groups that could then be sorted. An approach that worked well was to identify the groups by the values it "should" have within it. Each group could then be built by determining which values were candidates for that group. While this did lead to some values being placed into more than one group, the amount of overlap appeared small and once each group was fully ordered, it was easy to discard the values that did not belong.

With the algorithm implemented, I again took p to be the exact value specified in the paper. Not surprisingly, using this value for p caused the rounds to do an uneven amount of work. As in the case of median finding, I varied a multiplicative constant factor (Cp) between 0.2 and 0.4 in an attempt to find an ideal one.

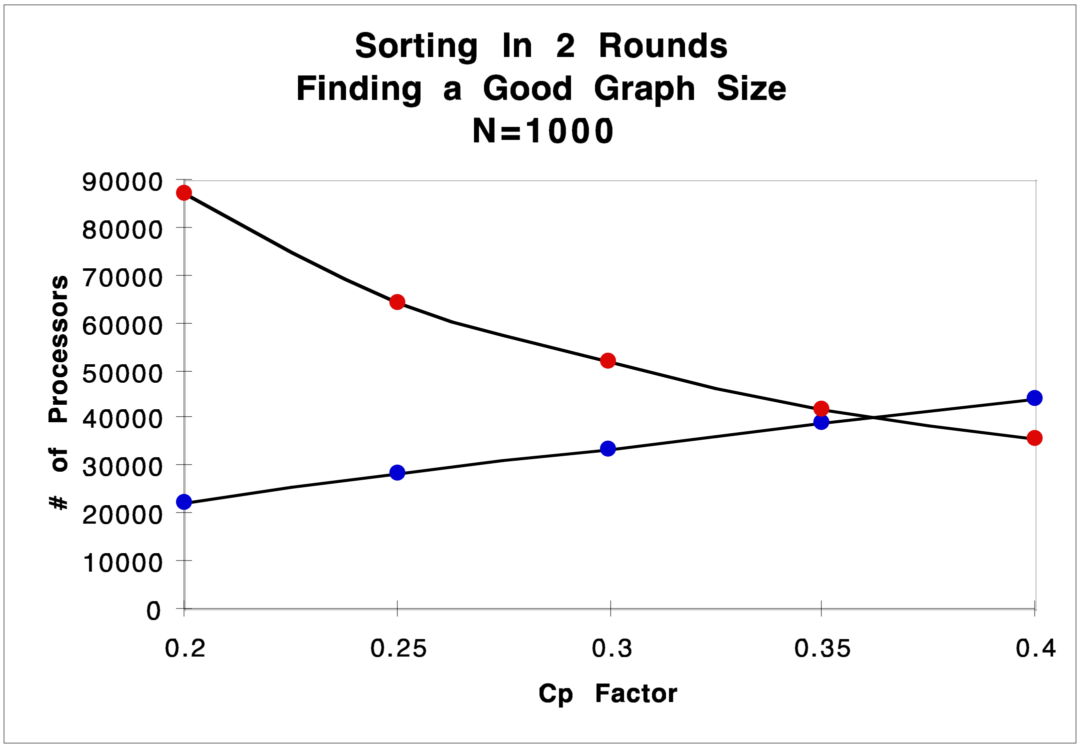

Figure 1.6

What I found was that when the graphs were generated with each edge being placed with a probability of .36*p* (Cp=.36), the first and second rounds used roughly the same number of processors. With this factor chosen, I went on to run more tests on a range of values for *n* and chart these results against the predicted requirements for 2 rounds of $O\left(n^{\frac{3}{2}}(\log n)\right)$ processors.

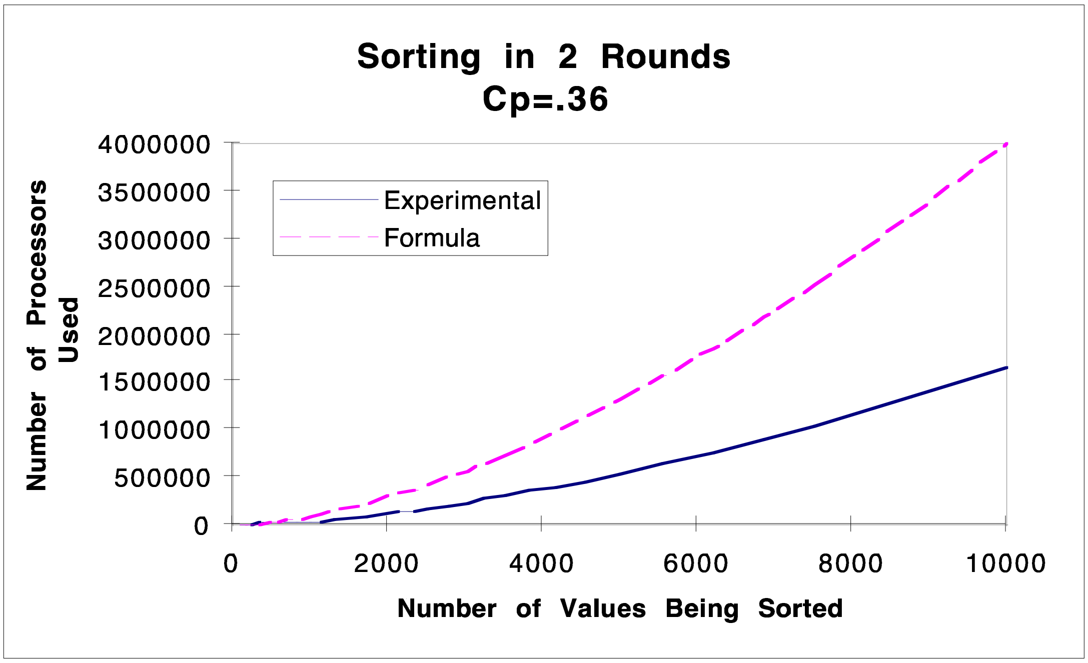

Figure 1.7

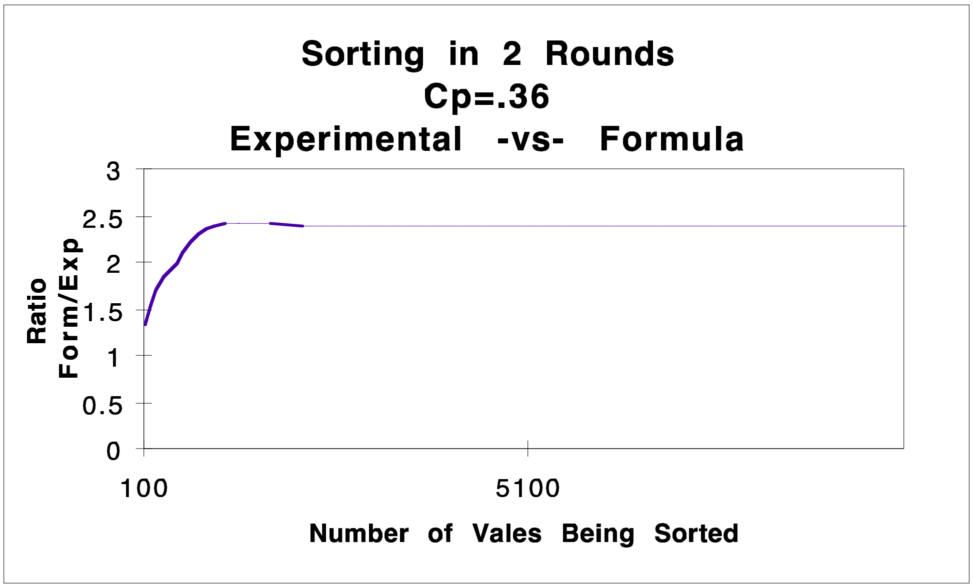

Figure 1.8

Figures 1.7 and 1.8 show the predicted required number of processors and the actual number used in the simulations. Figure 1.7 shows the number of processors used as n increases. Figure 1.8 shows the relative growth rate as n increases. The simulated results (based on the maximum of Round 1 or Round 2 over repeated trials) showed that the number of processors used has the same growth rate as the formula. We can use these empirical results to demonstrate that the growth rate predicted by the formula is accurate.

With these results affirming the validity of my implementation, I moved forward to implement a general k round algorithm. I implemented the k round algorithm by making the number of rounds to use a parameter of the function and then having it call itself (with the number of round left reduced by one) recursively on the individual groups. I experimented with the value of Cp and found that using .36 for the a-expanding graphs generated at the 3 round level worked well. Since the probability p was not based on the number of rounds or any other factor specific to the use of the a-expanding graphs, it is reasonable to have found a single constant factor that works well regardless of how many rounds are being used.

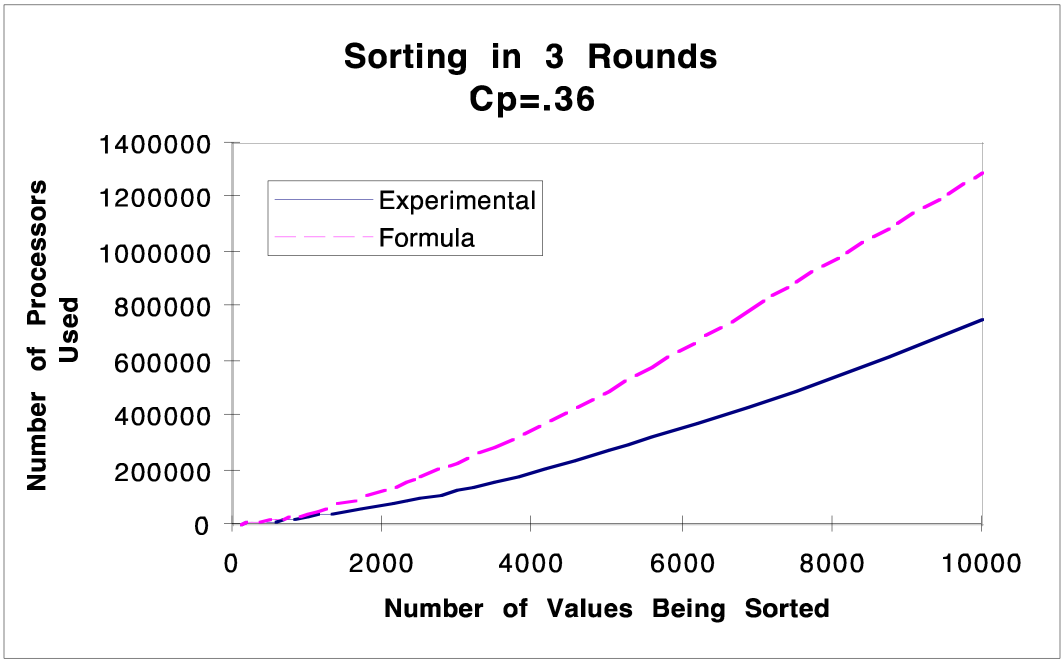

Figure 1.9

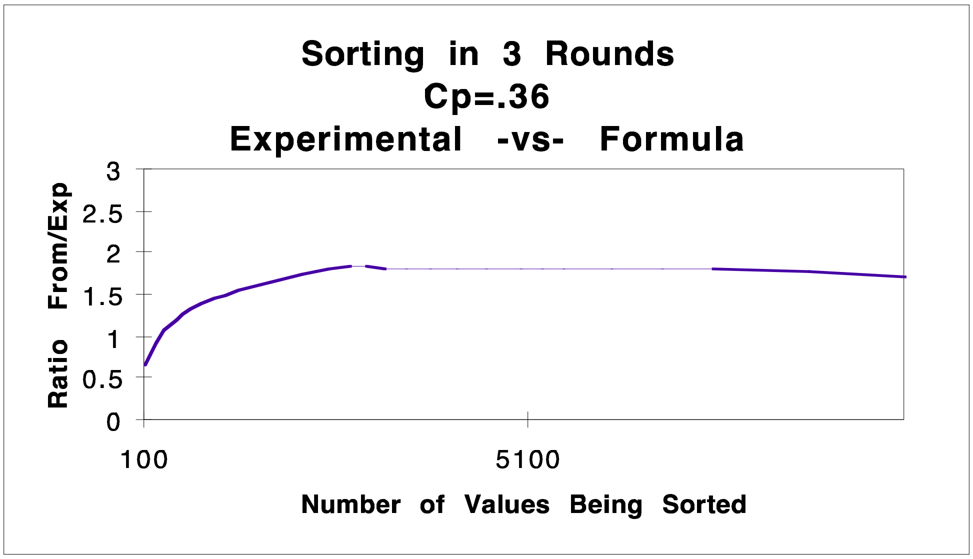

Figure 1.10

Figure 1.10 shows that once n starts to get large (around 2000), the relative growth rate of the number of processors required (empirical results versus the formula derived from the proof) is nearly constant.

I then continued to test the results for 4 and 5 round sorting, assuming that .36 was a good Cp factor to use in all sorting cases. The results showed that the number of processors required decreased in accordance with the given formula.

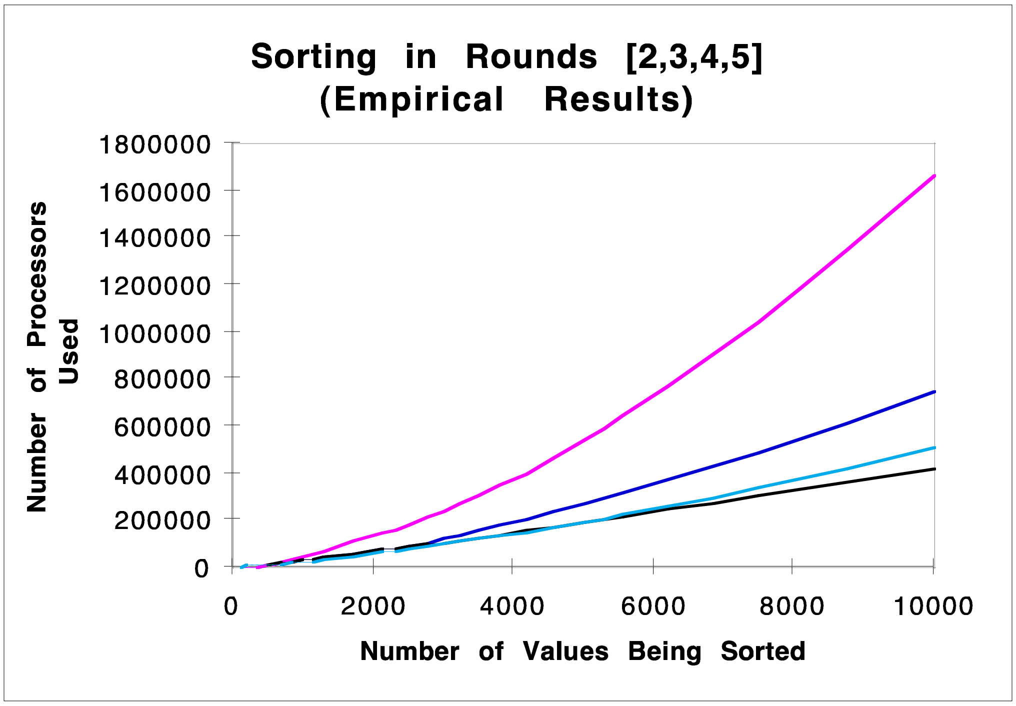

Figure 1.11

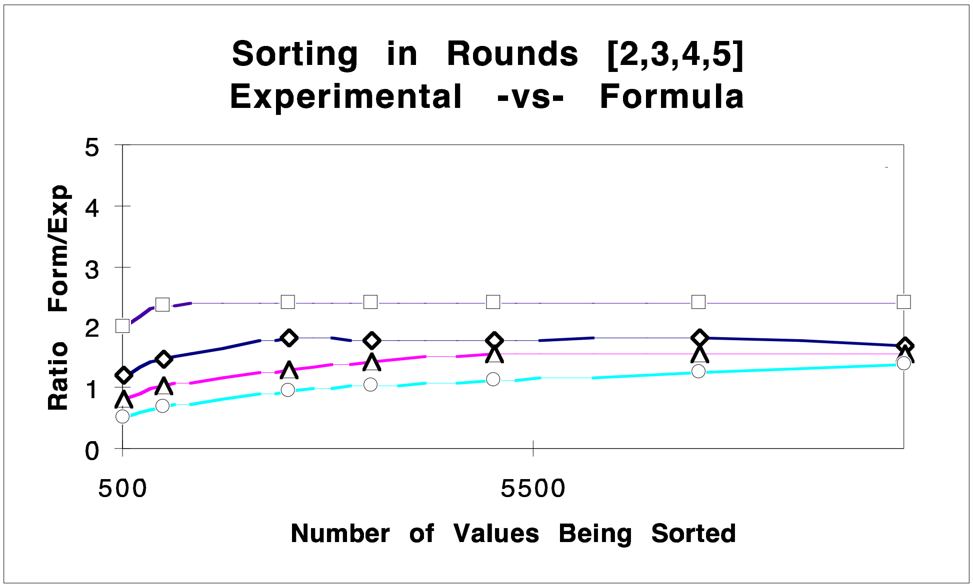

Figure 1.12

Figure 1.12 shows that for each of the cases which were simulated, once *n* starts to get large, the relative growth rate of the number of processors required (empirical results versus the formula derived from the proof) is nearly constant.

These experiences have shown that Pippenger's non-constructive algorithm can be used to create a computer usable implementation of the algorithm. Additionally, through empirical experimentation, it was shown that smaller graphs (nearly one third the size) could be constructed that would still greatly reduce the number of processors needed in later rounds.

Section 1.3: Investigating impact of block size

I considered experimenting with the size of the groups. The paper said that groups of size $O\left(n^{1-\frac{1}{k}}(\log n)^{\frac{2}{k}}\right)$ would exist. As I read the paper, I noticed that it was possible that using the same exact size for all of the groups would not work well. My first thought was to build the groups in a way that no two groups would share any members. I quickly found that this would be difficult if not impossible to do. The values are only partially ordered (for any element we know there are some number of elements below it and above it, giving it a range in which it belongs) and the algorithm does not guarantee exact dividing points between groups. With this, I had the program create sets of candidates for each group. Each set of candidates was slightly larger than the size of the group, but as the group sizes grew, this overflow (and in turn overlap of work) did not appear to grow. After the preliminary results matched the growth rate of the formula, I varied the size of the groups, but found that using exactly $n^{1-\frac{1}{k}}(\log n)^{\frac{2}{k}}$ for the group size worked best.

The final thing that was highlighted by these experiments was the cost of taking the transitive closure of the ordering information between rounds. The work done here would also be spread across all processors being used. While this work done between rounds is not considered by the literature when discussing the run time of the various parallel algorithms, it might be advantageous to use slightly more processors if this transitive closure work could be reduced. The real time overhead of doing transitive closure while simulating a massively multiprocessor machine on a single processor also made experiments on larger values of n prohibitive.

Section 1.4: Are the graphs being used really *a*-expander graphs?

While the graphs that were generated using the stated probability p did perform well, two questions exist: are they really a-expander graphs and can a smaller formula for p be used to still obtain graphs that work well.

There are techniques to test whether a given graph is an a-expanding graph. However, even if a graph is not a perfect a-expander graph, it may still perform well in the algorithm. As the probability p was being fine tuned, we saw that the number of questions (or comparisons) remaining for later rounds did change. In the following graph we see that as the constant factor applied to p is increased from .01 to .60 there is at first a steep drop off in the number of remaining questions, and then a more gradual decrease.

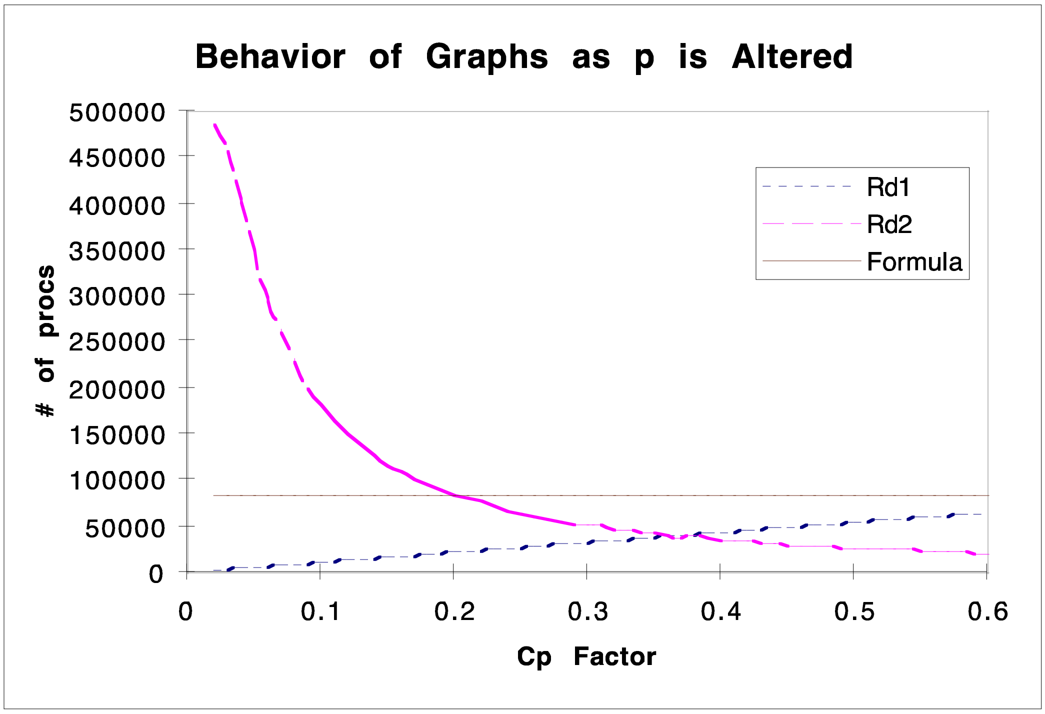

Figure 1.11

The point at which the decrease becomes more gradual represents the point at which we begin to get “good” graphs. These graphs might not technically be a-expanders, but they possess enough of the associated properties to perform well in their given task. These results appear to correspond to the findings of [BT2] which discuss how when building certain types of graphs, an absolute breakpoint can be found before which the graphs are not useful but after which they are.

Section 1.5: Summary of empirical results

Through these experiments I can address both of the philosophies that I presented earlier:

<u>Philosophy 1: Non-constructive proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.</u>

I have shown that this non-constructive proof can be used as the basis for the design of a computer program that would in fact sort a given set of values in a given number of rounds using the stated number of processors. There is some preprocessing required (specifically the generation and confirmation of the a-expanding graph), but this would only be required once for a given input size. Once a suitable graph has been generated, it can be reused. The generation of a usable graph with the given factor of p should be accomplished quickly (polynomial time) as it is likely that the first graph generated will work well in practice.

<u>Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.</u>

In these empirical studies, the value of p was fine tuned in a way that was not predicted by Pippenger's proof. Additionally, the performance of the graphs as the constant factor was changed implies that there is a breakpoint before which graphs that are not quite a-expanders do not work well and after which they do.

While this might be obtainable by extending previous theoretical results in other areas [BT2] it is interesting to observe the actual performance here. The empirical results also showed that there was not a large constant being hidden in the order notation when discussing the number of porcessors which would be required.

Chapter 2 : Häggkvist and Hell's K-Round Sorting Algorithm

Section 2.1: Sketch of algorithm

Häggkvist and Hell [HH2] present constructive proofs for the following upper bounds :

$$\mathrm{Merge}(n,k) = \Theta\left(n^{\frac{2^k}{2^k-1}}\right)$$

$$\mathrm{Sort}(n,k) = O\left(n^{1+\frac{2}{\sqrt{2k}}}\right)$$

The sorting algorithm heavily depends upon the use of parallel merging. As a result of this, merging itself became an object of interest in the paper. The paper gives matching upper and lower bounds for merging. While all that was needed was an upper bound for merging, knowing the exact bound allows us to know that the sorting algorithm cannot be improved upon via an improvement to the bound on merging.

The algorithm given for merging two ordered lists of n elements is to partition each list into groups, and then do a pairwise comparison of the first element of each group in the first list with the first element of each group in the second list. After doing these comparisons, there will be a (small) limited number of groups whose members are still unordered relative to one another. The proof of this builds on the fact that if you cluster the elements into nodes of a graph in a certain way, and only add edges between two cluster nodes if there is an uncompared pair of elements across the two clusters, this graph is planar (and thus linear in size).

Theorem 2.1[HH2]: Merging in k rounds can be done with $O\left(n^{\frac{2^k}{2^k-1}}\right)$ processors

Häggkvist and Hell establish that a group size of $O\left(n^{\frac{1}{3}}\right)$ is optimal for two round parallel merging, giving Merge(n,2)=$O\left(n^{\frac{4}{3}}\right)$. By applying inductive methods on the merging of the groups whose orientation was not previously determined by the comparison of the first elements of each group, they derive the generalization Merge(n,k) = $O\left(n^{\frac{2^k}{2^k-1}}\right)$. ♦

Theorem 2.2[HH2]: Merging in k rounds requires $\Omega\left(n^{\frac{2^k}{2^k-1}}\right)$ processors

As with the upper bound proof, the lower bound proof begins with a proof on two rounds and is then extended by induction. The following is a sketch of Häggkvist and Hell's proof that merging in 2 rounds requires $\Omega\left(n^{\frac{4}{3}}\right)$ processors:

- Assume you could merge two lists in two rounds with fewer than $\frac{1}{4}n^{\frac{4}{3}}$ processors.
- Let G be the bipartite graph representing the first round of comparisons.
- Build G' representing clusters of uncompared pairs after that first round and show that there are more than $\frac{1}{4}n^{\frac{4}{3}}$ comparisons remaining.

This can then be extended through induction to the result of

Merge(n,k) = $\Omega\left(n^{\frac{2^k}{2^k-1}}\right)$. ♦

<u>Theorem 2.3[HH2]: Sorting in 3 rounds can be done with $O\left(n^{\frac{8}{5}}\right)$ processors</u>

The algorithm to sort a list of values in *k* rounds is based on using some number of rounds (*j*) to partition the list and sort each partition, and then use the remaining *k-j* rounds to do a pairwise merge of those partitions. In the 3 round case, the list is partitioned into $n^{\frac{2}{5}}$ groups of size $n^{\frac{3}{5}}$ and each partition is then sorted in one round using $n^{\frac{6}{5}}$ processors per partition, or a total of $n^{\frac{8}{5}}$ processors. Then in the two remaining rounds, a pairwise merging of the $n^{\frac{2}{5}}$ groups would produce all information required to fully order the original n values. ♦

They give a set of formulas that can be used to determine how many rounds to use on each stage (sorting/merging) based on the number of rounds that you have available. They then go on to give estimates based on this information for three, four and five round merging.

$$\mathrm{Sort}(n,3) = \mathrm{O}\left(n^{\frac{8}{5}}\right)$$

$$\mathrm{Sort}(n,4) = \mathrm{O}\left(n^{\frac{20}{13}}\right)$$

$$\mathrm{Sort}(n,5) = \mathrm{O}\left(n^{\frac{28}{19}}\right)$$

Each of these assumes the use of one round to sort the partitions and then k-1 rounds to merge those partitions. They also assume that the partitions are all of the same size. If the original list does not partition evenly, then the last partition can be padded out to match the size of the others. A general formula is not given since it is actually a recurrence which involves taking the best sort/merge split recursively. An approximation of this is given as $O\left(n^{1+\frac{2}{\sqrt{2k}}}\right)$.

Section 2.2: Implementation of algorithm

With this information, I went on to implement 3 round sorting. From the algorithm you can derive the fact that you should first partition the original list into groups of size $O\left(n^{\frac{3}{5}}\right)$ and then do a two round pairwise merge across these partitions. The paper explicitly said the groups should be of size $O\left(n^{\frac{1}{3}}\right)$ for two round merging. I began by partitioning into lists exactly of size $n^{\frac{3}{5}}$ (m) and then doing a pairwise merge using groups exactly of size $m^{\frac{1}{3}}$. The results when I tested my program on values of *n* up to 2000 showed that the second and third rounds (the ones being used to merge) were both using less than $n^{\frac{8}{5}}$ processors. However, there was a difference in the number of processors used in each round.

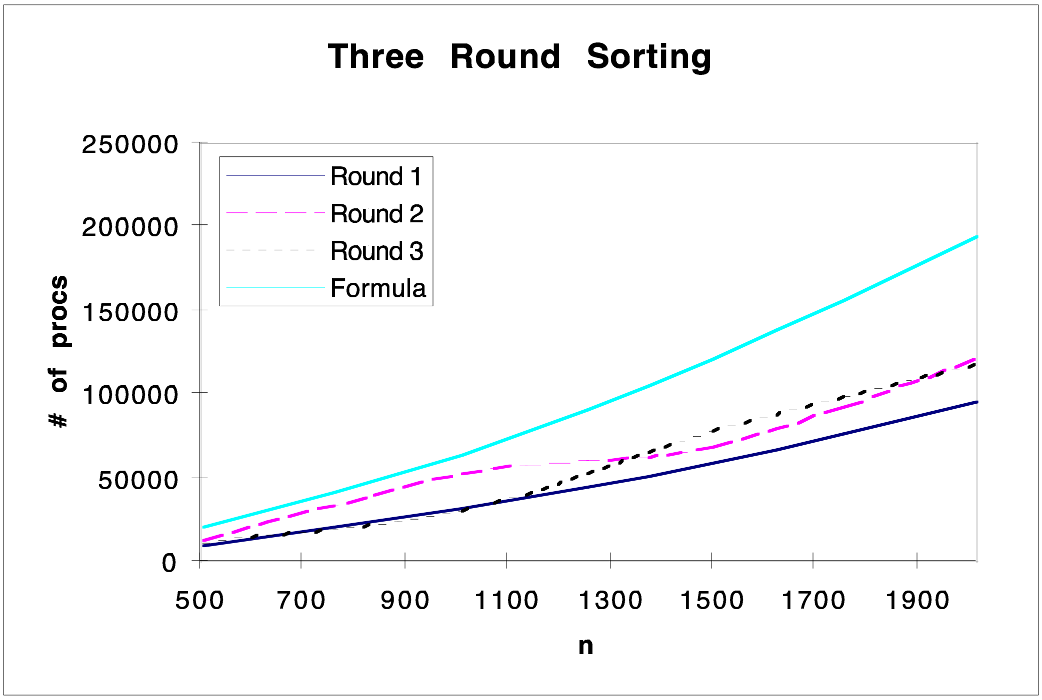

Figure 2.4

To see if that difference would disappear as the value of *n* increased, I extended testing to values of *n* up to 10000.

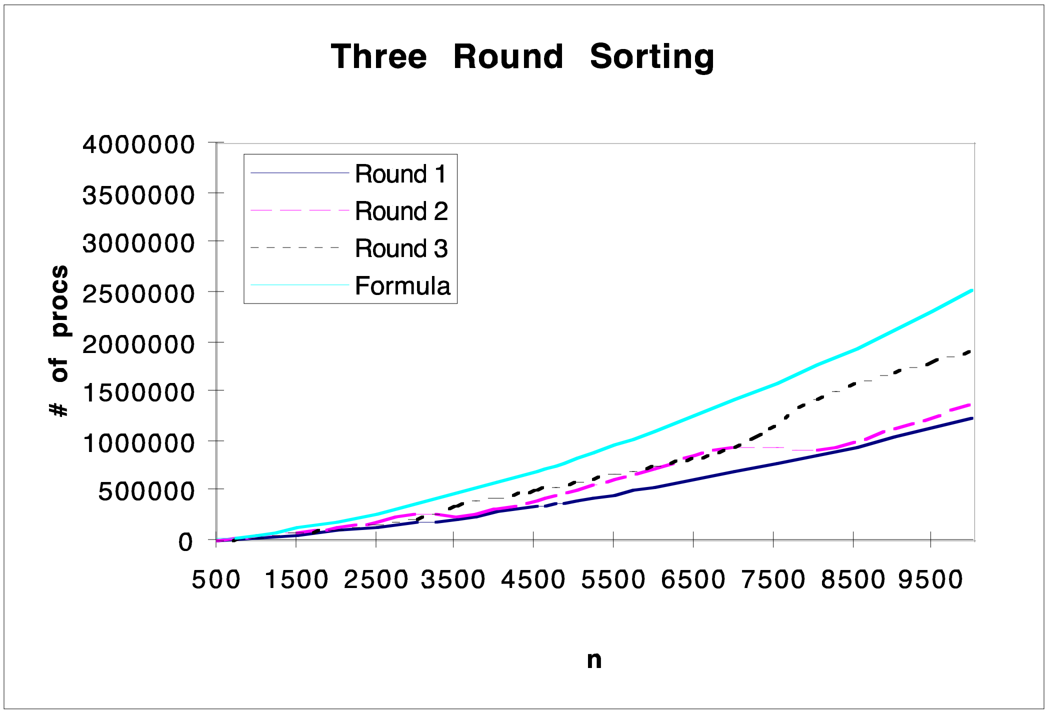

Figure 2.5

The chart shows that the difference remained. This could have indicated that while the growth rate is following the formula of $n^{\frac{8}{5}}$, some improvement might have been attainable by bring the number of processors used in rounds two and three closer together. However, by varying the size of the partitions as well as the size of the groups during merging by a constant factor, I found that small changes did not significantly effect the end result and that large changes had detrimental effects. Additionally, the points at which this occurred were not constant. The following charts demonstrate these differences on several values of *n*.

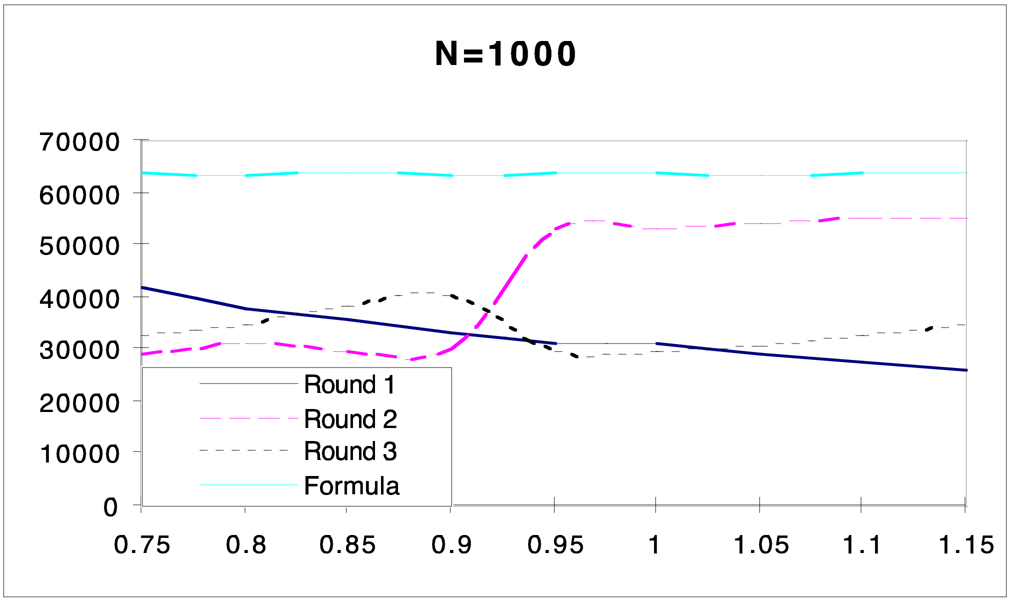

Figure 2.6

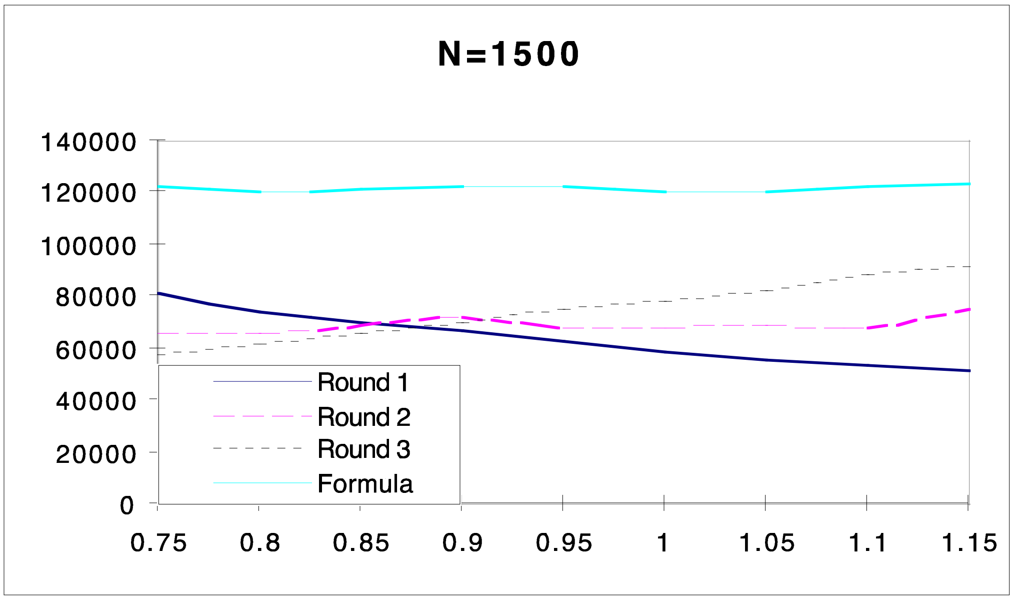

Figure 2.7

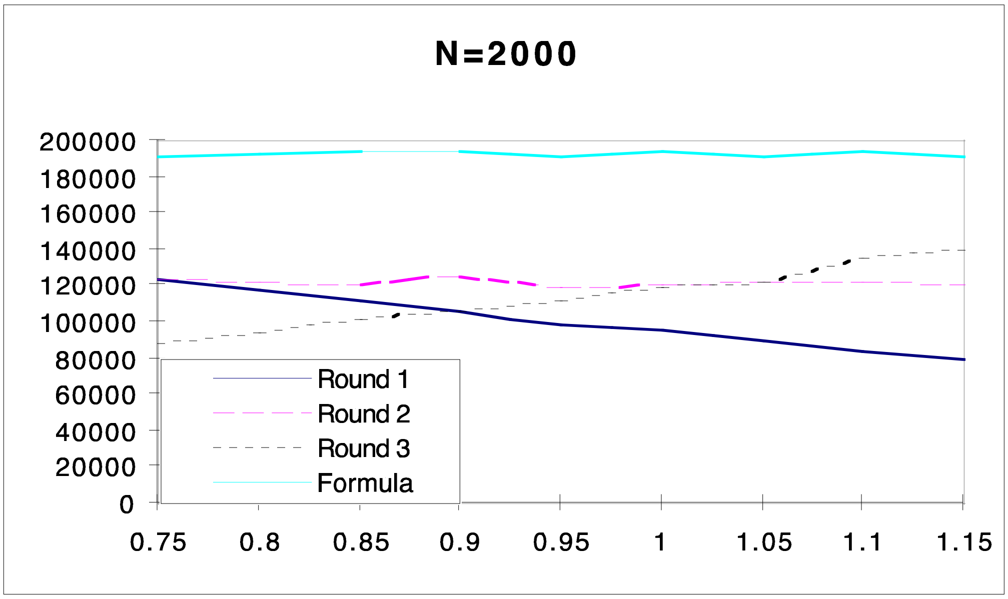

Figure 2.8

While it was possible in some cases to bring the number of processors used in these rounds closer together, on the whole, using the initial values proved best.

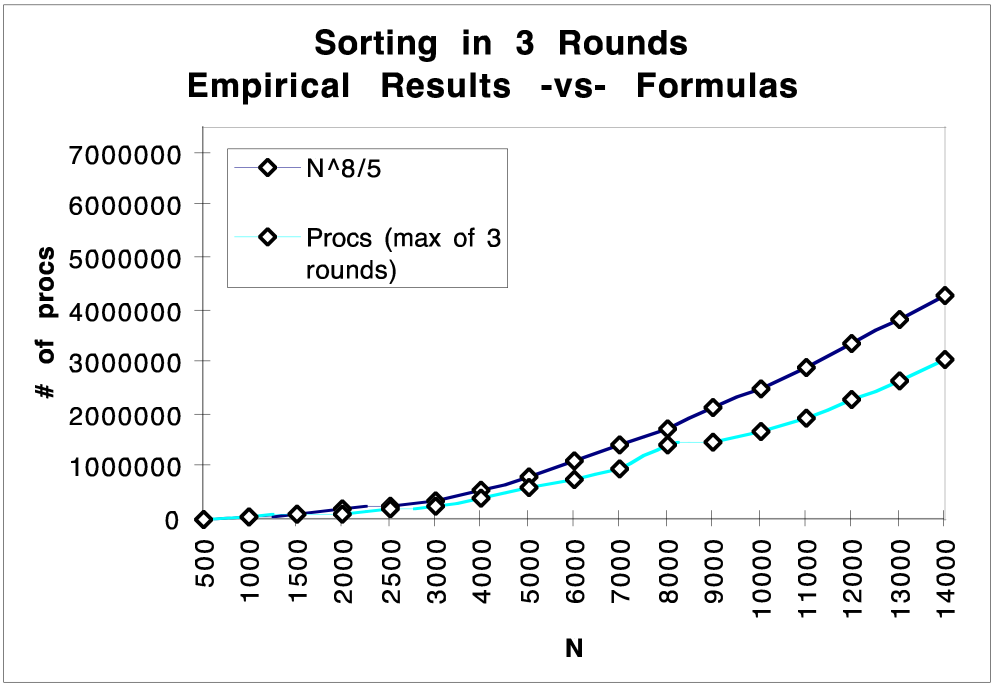

Figure 2.9

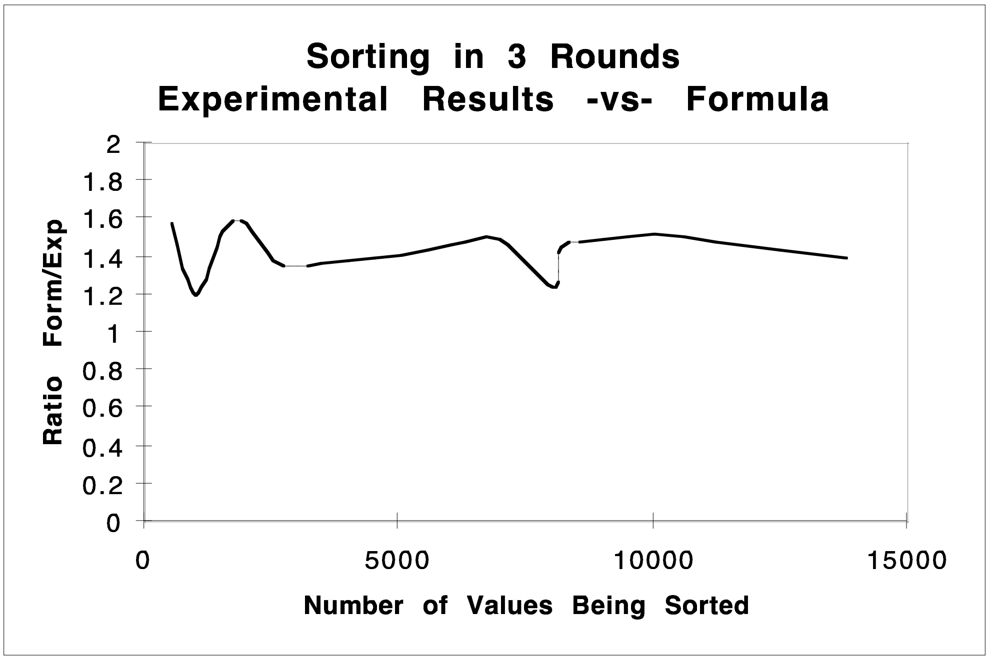

Figure 2.10

Figure 2.10 shows that the relative growth rate between the formula's predictions and the empirical results as n increases is pretty much constant.

To move from three round sorting to four round sorting, the only part of the algorithm that changed was the merging. Instead of using two rounds to merge, the program could now use three rounds to do the pairwise merging of the partitioned list. Now, after the first elements of each group had been compared to one another, the groups whose positions were not known could be merged using the two round algorithm.

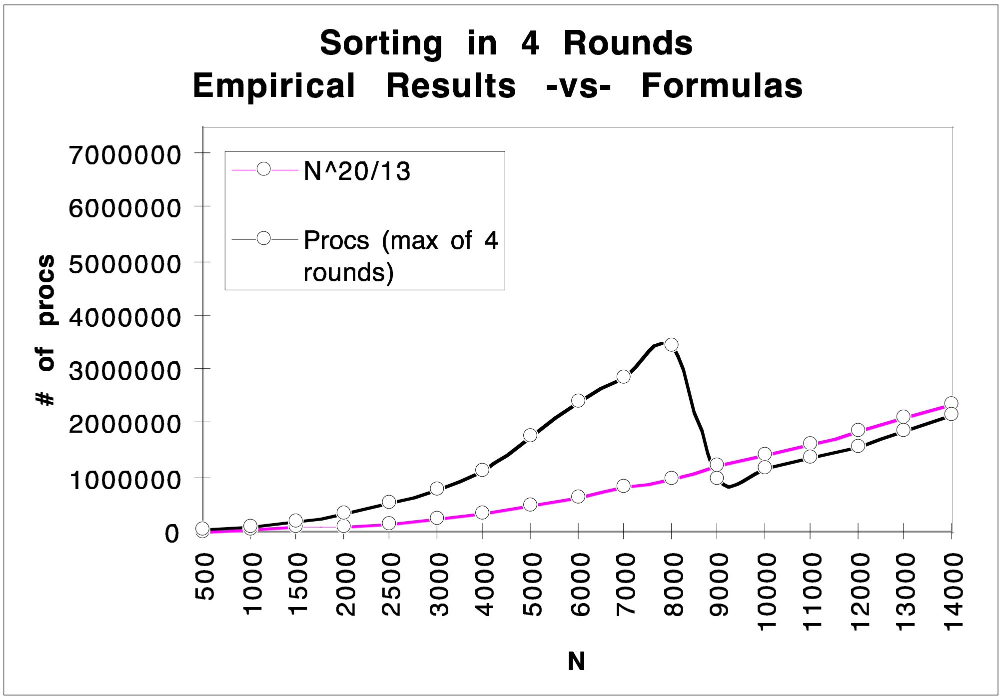

Figure 2.11

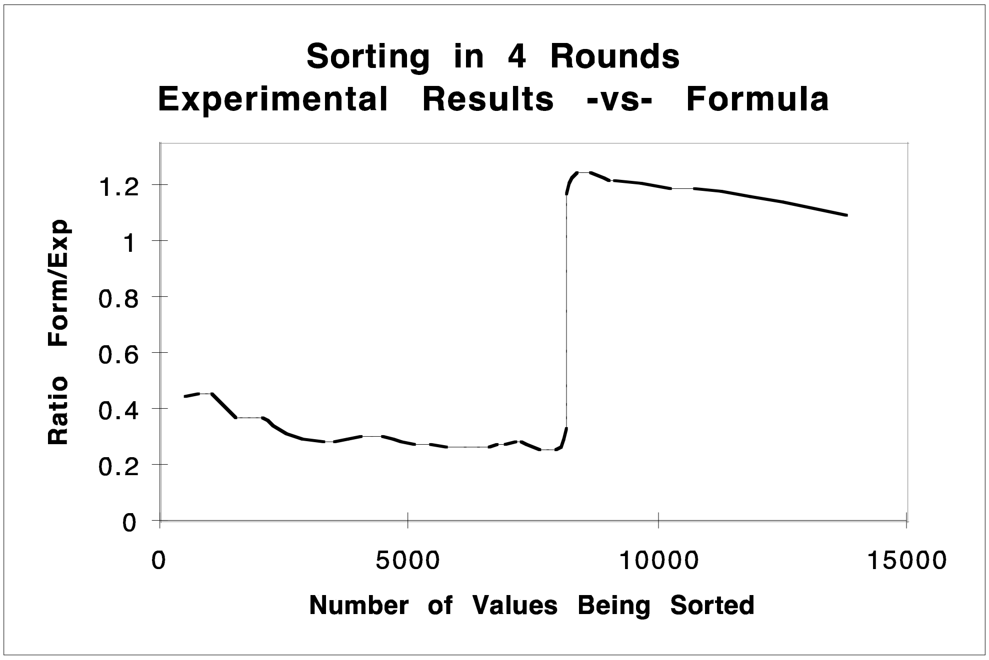

Figure 2.12

Figures 2.11 and 2.12 show that until n reaches 8192 the algorithm is requiring more processors that predicted, but that once 8192 is reached it gives the results that it promised. The value of 8192 is (not coincidentally) 2^{13}. It is at this point that the groups are created in such a way that work is actually done in the fourth round. Once this mark is reached, the four round sorting algorithm performs as predicted by the formula of $O\left(n^{\frac{20}{13}}\right)$. Due to these results, I did not continue on to do empirical tests on sorting in five rounds since n would need to be at least 2^{19} before useful results would be obtained. Further work could investigate generating a formula to predict where this cutoff would fall for a given number of rounds.

Section 2.3: Difficulties encountered during implementation

Implementing this algorithm on a sequential processor presented several challenges. One such challenge was deciding how to hold the information gathered. The premise is that all information required to order the original list would be generated by following the algorithm, but this information needs to be stored in a manner that could lead to the actually ordering. I decided to use an adjacency matrix that would be updated with the information obtained during each round. When merge(n,k) would call down to merge(m,k-1) (where m would be the size of each group in the k round merge) the information obtained would assume that the lists were of size m rather than n. In order to integrate the information obtained from these recursive calls, the information obtained in the local merges had to be re-oriented to be placed into the matrix at the previous calling level. I do not know if these issues would apply when implemented on an actual parallel machine.

Another challenge was dealing with the even partitioning at each level of merging. While padding the information would work, it appeared that the implementation would be made more difficult if the padding needed to be done internally at each level of merging. The solution I employed was to choose values for n that would partition well at each level. I was able to generate these values with a short program. The program could iterate through values of n close to the one that I wanted to use and print out that ones could be easily manipulated so they would partition evenly at each level.

A hidden cost within this sorting algorithm is the cost of propagating the information obtained during the merging rounds. After the first elements of each group are compared, the adjacency matrix needs to have all of the information that this generates filled in. This will essentially take the same amount of time as the transitive closure of the matrix. Additionally, after the remaining groups are merged, this information has to be incorporated into the primary matrix. While this second cost could possibly be avoided by a more clever use of the matrices, the first one can not be if our goal is to have an ordered list at the end.

Section 2.4: Summary of empirical results

Through these experiments I can address both of the philosophies that I presented earlier:

<u>Philosophy 1: Non-constructive proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.</u>

While this proof is a constructive one, the issues that arose during implementation were more complex than with Pippenger's. Due to the pairwise merging at every level and associated requirement that all lists being merged have the same length, there is a good deal of extra overhead required to coordinate this.

<u>Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.</u>

In addition to the overhead for coordinating the blocks at each level, these empirical studies led to the observation that n needs to be sufficiently large so that there are no empty blocks at the bottom of the recursion. While the paper does say that the algorithm works for "sufficiently large values of n" it did not indicate what this meant in terms of specific values of n. After doing the empirical tests, we now know that in the case of four round sorting, *n* must be at least 2^{13}. Additionally, by observing the algorithm in action I have ascertained the cause of this boundary and have extended this to the assumption that for five round sorting, *n* must be at least 2^{19}. The values for which *n* is too small to use this algorithm is something that may have been found through a more detailed analysis of the algorithm. However, that value and its origins were highlighted through the experimental results.

Chapter 3 : Alon's 2-Round Sorting Algorithm Using Direct Implications

Section 3.1: Sketch of algorithm

Alon [Alon1] presents a constructive proof of the following upper bound:

$$\text{Sort}\,(n,2) = O\left(n^{\frac{7}{4}}\right)$$

This algorithm, while slightly worse than current constructive upper bounds had the advantage of using only a limited form of transitive closure between rounds (specifically, only direct implications). Alon used techniques in projective geometry to construct graphs that could be used in the creation of such an algorithm. Additionally, it is less complicated to implement than those better algorithms. Pippenger's constructive algorithm uses Ramanujan graphs whose constructions are not straightforward. Wigderson and Zuckerman's algorithm will be discussed in Chapter 6.

<u>Definition 3.1: Direct implications</u>

Given a graph, we can add edges to the graph by adding connections based on direct implications. This means if in our original comparison graph we have

$x>y$ and $y>z$ then we will add to that graph the fact that $x>z$. Note that if we have $x>y$, $y>z$, $z>w$ we do **not** obtain $x>w$.

Definition 3.2: Geometric expanders

Geometric expanders are built using projective geometry over a finite field. An individual geometric expander is based upon two inputs, a prime number q and a dimension d. They are built as follows:

- Create a set of d+1 tuples of the following form

$$\left(1, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \ldots, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}\right)$$

$$\left(0, 1, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \ldots, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}\right)$$

$$\left(0, 0, 1, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}, \ldots, \begin{Bmatrix} 0 \\ \vdots \\ q\text{-}1 \end{Bmatrix}\right)$$

  etc.

  [NOTE: Each tuple represents a hyperplane in d+1 space over the finite field of q]

- In the bipartite graph G(U,V,E) that we are building, allow each tuple to represent corresponding vertices in U and V.
- An edge will exist between $u \in U$ and $v \in V$ in G if the planes which represent u and v are orthogonal to one another. That is $u \bullet v \equiv 0 \bmod q$.

Theorem 3.3[Alon1]: Sorting can be done in 2 rounds using only direct implication with $O\left(n^{\frac{7}{4}}\right)$ processors

There are two inputs when building geometric expanders [Alon1], a prime power q and a dimension d. The number of vertices in each half of this type of bipartite graph is $\frac{q^{d+1}-1}{q-1}$. The degree of any given vertex in this type of graph is $\frac{q^{d}-1}{q-1}$. If you set $n = \frac{q^{d+1}-1}{q-1}$ then we can say that $q = O\left(n^{\frac{1}{d}}\right)$ and that the degree is $O\left(n^{1-\frac{1}{d}}\right)$. Finally, we know that graph will have $O\left(n^{2-\frac{1}{d}}\right)$ edges. Alon used graphs of this type with dimension 4 and showed that due to their expanding qualities, if a bipartite graph (x_1 to x_n on each side of the graph) of this type is used to specifiy the comparisons done in the first round of a two round sorting algorithm and only direct implications were computed between rounds that there would only be $O\left(n^{\frac{7}{4}}\right)$ comparisons remaining for the second round. Since there are $O\left(n^{\frac{7}{4}}\right)$ edges in this type of graph, this gives an $O\left(n^{\frac{7}{4}}\right)$ two round sorting algorithm that only requires direct implications between rounds. NOTE: We will elaborate on this proof later. ♦

Section 3.2: Implementation of Alon's Algorithm

In order to implement this algorithm, I needed to be able to build the specified geometric expanders and then use them to direct the first round's comparisons. Between rounds I needed to deduce any direct implications from that set of comparisons. For the second round I needed to execute all remaining unanswered comparisons.

The results of these experiments (see Figure 3.4) showed that while the first round (as specified by the graph) did take $O\left(n^{\frac{7}{4}}\right)$ processors, after doing the partial transitive closure (PTC), there were far fewer comparisons remaining than expected. The explanation of this phenomena is left as an open question.

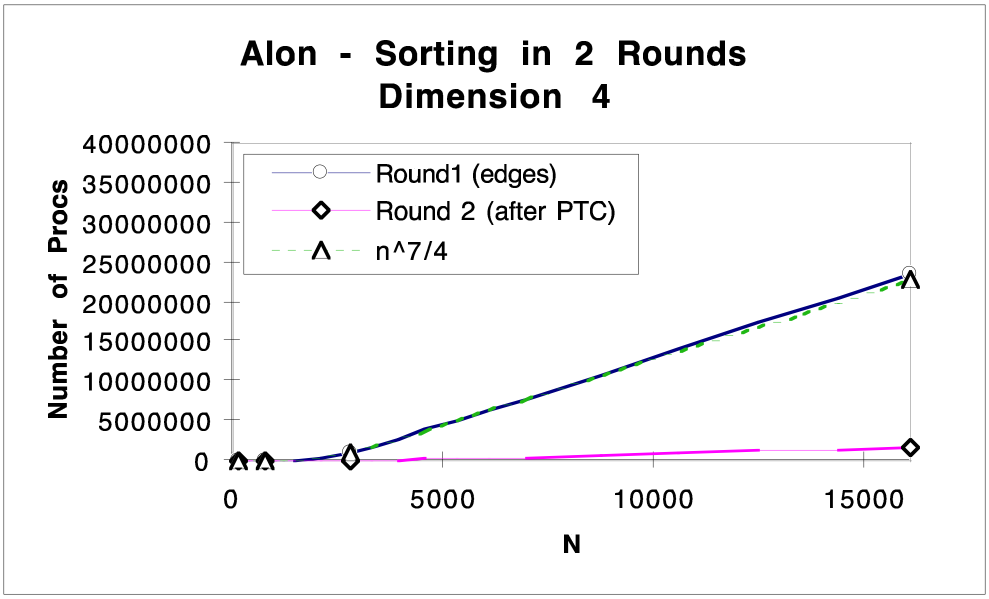

Figure 3.4

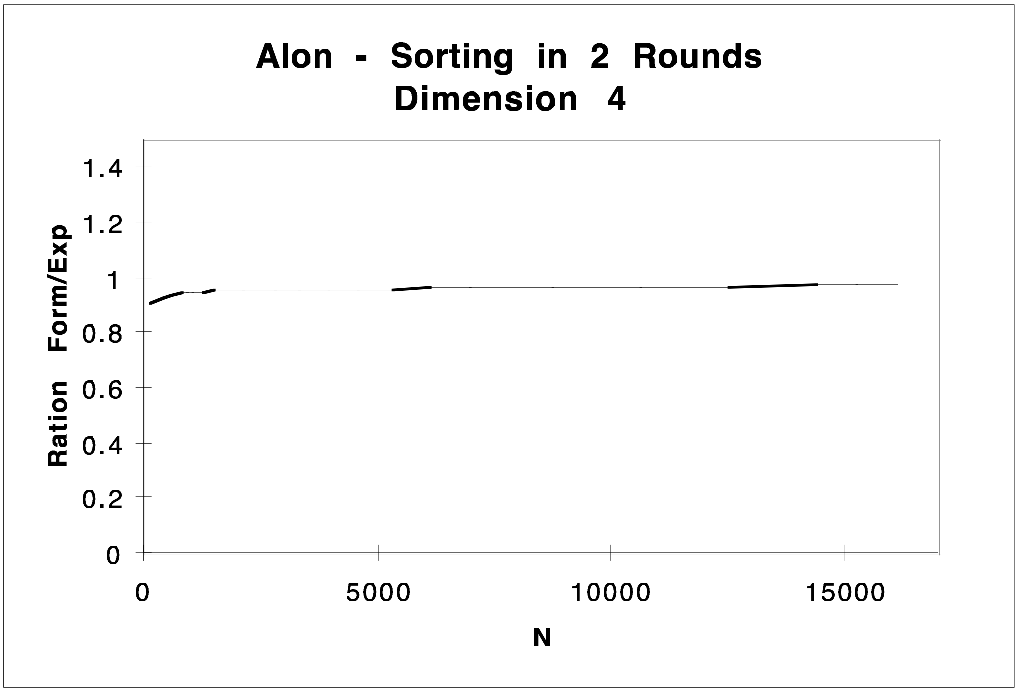

Figure 3.5

Upon seeing this I experimented with block sizes as I had in the case of the [HH2] algorithm. However, this did not appear to have any real impact. A new variable available for manipulation in this algorithm was the dimension. I experimented with changing the geometric expander so that it was of dimension 2 or 3 rather than 4. This led to some interesting results.

Using dimension 3 appeared to give a noticable improvement over the dimension 4 case.

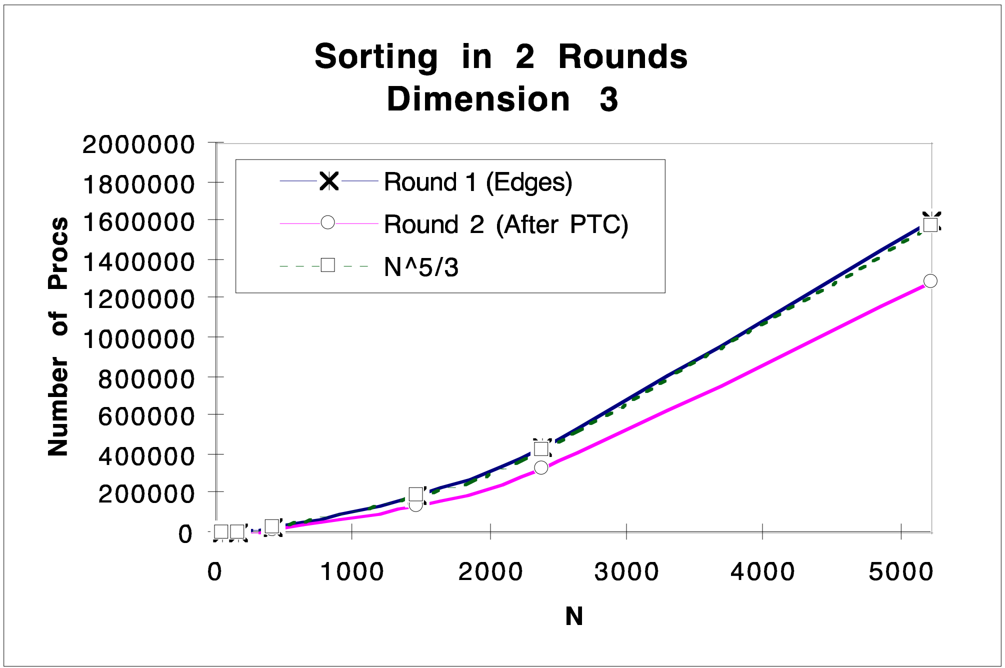

Figure 3.6

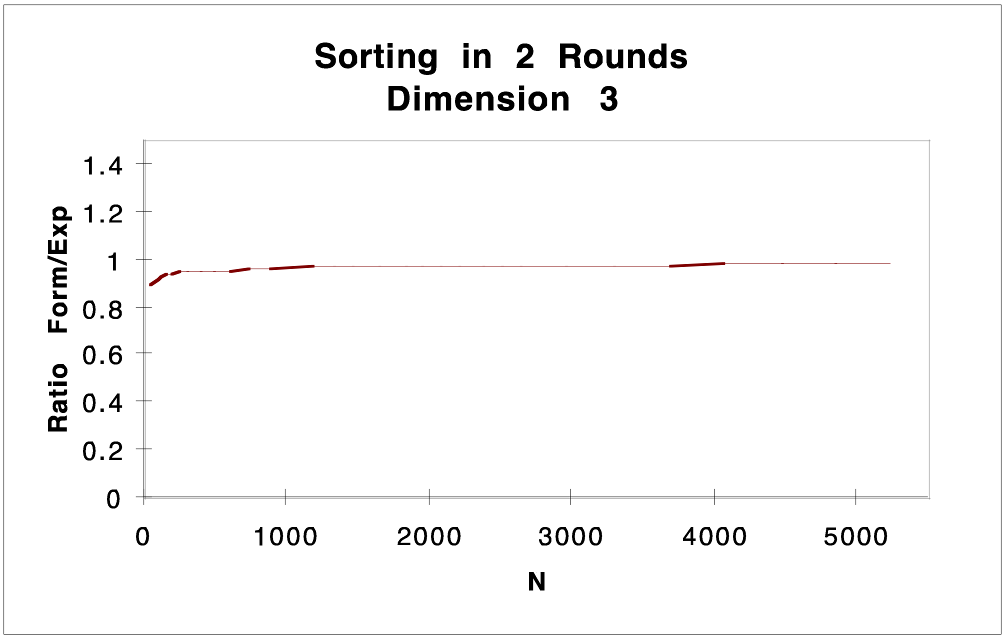

Figure 3.7

The number of processors used in the first round is $O\left(n^{\frac{5}{3}}\right)$ as would be expected using geometric expanders of dimension 3, but the remaining comparisons in the second round also appeared to be $O\left(n^{\frac{5}{3}}\right)$. The known lower bound from [BT1] is $\Omega\left(n^{\frac{5}{3}}\right)$.

Section 3.3: Some new thoughts on the algorithm and the proof

<u>Definition 3.8:</u> $(\alpha n^a, \beta n^b, \chi n^c, \delta n^d)$-expanders

A bipartite graph on (U,V,E) with $|U| = |V| = n$ will be called an $(\alpha n^a, \beta n^b, \chi n^c, \delta n^d)$-expander if it has the following two properties:

1) $(\forall Z \subseteq V)\left[|Z| \geq \alpha n^a \Rightarrow \left|\left\{x \in U : |N(x) \cap Z| \leq \beta n^b\right\}\right| \leq \chi n^c\right]$

2) $(\forall Y \subseteq V)\left[|Y| \geq \beta n^b \Rightarrow |N(Y)| \geq n - \delta n^d\right]$ ♦

The special case of Theorem 3.9 (below) with $\left(3n^{\frac{3}{4}}, n^{\frac{1}{2}}, n^{\frac{1}{2}}, n^{\frac{3}{4}}\right)$-expanders of size $O\left(n^{\frac{7}{4}}\right)$ is implicit in [Alon1].

<u>Theorem 3.9: If there exist $(\alpha n^a, \beta n^b, \chi n^c, \delta n^d)$-expanders of size $O(n^e)$, then we can sort in 2 rounds using only direct implications with $O\left(n^{\max(e, d+1, c+2-a, a+1)}\right)$ processors.</u>

Assume that we have a set of n values that we want to order and an $(\alpha n^a, \beta n^b, \chi n^c, \delta n^d)$-expanding graph of size $O(n^e)$ connecting n vertices representing the n values.

For Round 1, we do the comparisons as designated by the graph. This requires $O\left(n^e\right)$ processors. (NOTE: We will be using a bipartite graph on $2n$ vertices during our construction, but since it is a symmetric bipartite graph, we can view it as if an n vertex graph were being used when we actually sort by folding the two halves onto one another.)

Since we know that there exists an ordering of the n values, we can imagine having the vertices correctly ordered. Note that the graph connecting these vertices is still an $\left(\alpha n^a, \beta n^b, \chi n^c, \delta n^d\right)$-expander. We can then divide the vertices into $\frac{1}{\alpha} n^{1-a}$ blocks $(A_1, A_2, \ldots, A_{\frac{1}{\alpha} n^{1-a}})$ each of αn^a vertices.

We are interested in knowing how many comparisons remain for Round 2. Take an ordered pair (x,y) with $x \in A_{i+1}$ and $y \in A_1 \cup \cdots \cup A_{i-1}$. Set $Z = A_i$. By Part 1 of Definition 3.8, $\left|\left\{x \in U : |N(x) \cap Z| \le \beta n^b\right\}\right| \le \chi n^c$.

i) Look at x not in this set

Those x have at least βn^b neighbors in A_i. By Part 2 of Definition 3.8, since $|N(x)| \ge \beta n^b$ we know that $N(N(x)) \ge n - \delta n^d$. That leaves $\delta n^d \alpha n^a$ uncompared pairs per block for Round 2. This contributes $\frac{1}{\alpha} n^{1-a} \delta n^d \alpha n^a$ or $O\left(n^{d+1}\right)$ comparisons to Round 2.

ii) Look at x in the set

In the worst case we would need to compare each of these χn^c values to all other (n-1) values. This contributes $\frac{1}{\alpha} n^{1-a} \chi n^c (n-1)$ or $\mathrm{O}\left(n^{c+2-a}\right)$ possible comparisons to Round 2.

iii) We also have to look within each block for possible values for y which have not been compared to x.

At most this will be $C(\alpha n^a, 2)$ which is $\mathrm{O}\left(n^{2a}\right)$. This contributes $\frac{1}{\alpha} n^{1-a} n^{2a} = \mathrm{O}\left(n^{a+1}\right)$ comparisons to Round 2.

This shows that $\mathrm{O}\left(n^{\max(e,d+1,c+2-a,a+1)}\right)$ processors will be required to sort n values given an $\left(\alpha n^a, \beta n^b, \chi n^c, \delta n^d\right)$-expander of size $\mathrm{O}\left(n^e\right)$. ♦

<u>Theorem 3.10 [Alon1]:</u>

(Theorem 2.1 in [Alon1])

Let G=(U,V,E) be a bipartite graph.

- Let $|U|$=n and $|V|$=m

- Let the degree of $u \in U = k$ and the degree of $v \in V = s$
- Let the adjacency matrix A be defined as

  $(a_{uv}) u \in U, v \in V$

  a_{uv}={1 if u and v are adjacent, 0 otherwise}
- $ks = \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ as the eigenvalues of AA^T

If you take $Z \subseteq V$, $|Z|=z$ and let $X=\{u \in U : |N(u) \cap Z| \leq b\}$ with $x=|X|$ then

$x(n(k-b)^2 - (m-z)(ks-\lambda_2)) \leq \lambda_2 n(m-z)$.

Furthermore, if $b \leq \frac{kz}{2m}$ the inequality reduces to:

$$x \leq \frac{\lambda_2 n(m-z)}{(n(k-b)^2 - (m-z)(ks-\lambda_2))} \quad \blacklozenge$$

The above theorem relies on $b \leq \frac{kz}{2m}$ so that x can be isolated by dividing both sides by $(n(k-b)^2 - (m-z)(ks-\lambda_2))$ without the danger of dividing by a negative value and thus switching the direction of our inequality. Since we want b to be small, and the value which is used for b in the $O\left(n^{\frac{7}{4}}\right)$ algorithm is at most $\frac{kz}{2m}$ there might not appear to be a reason to look for other values of b for which the above holds as well. However, to obtain the $O\left(n^{\frac{5}{3}}\right)$ algorithm, we would need to take $b > \frac{kz}{2m}$. For this reason, this proof can not be extended to explain our empirical results with dimension 3 geometric expanders. Additionally, we would need to take $b>2k$ to extend this formula to dimension 3. This would cause the mathematics to fail, but more importantly, it would not make sense to have $b>k$. If b were greater than k, then we would be talking about all vertices since each vertex is known to have k neighbors ($k<2k<b$).

Theorem 3.11 [Alon1]: some properties of geometric expanders

Theorem 2.3 in [Alon1] gives us certain information about geometric expanders with any given q and d. We will use n to represent the number of vertices in each half of this bipartite graph and k to represent the degree of any vertex.

If G is a geometric expnader on q and d then:

- $n = \frac{q^{d+1} - 1}{q - 1} = q^d + \Theta\left(q^{d-1}\right)$
- $k = \frac{q^d - 1}{q - 1} = q^{d-1} + \Theta\left(q^{d-2}\right) = n^{\frac{d-1}{d}} + \Theta\left(n^{\frac{d-2}{d}}\right)$
- G has $(1 + o(1))n^{2-\frac{1}{d}}$ edges

- $\lambda_2 = q^{d-1} = n^{\frac{d-1}{d}} + \Theta\left(n^{\frac{d-2}{d}}\right)$
- $|X| \geq x \Rightarrow |N(X)| \geq n - \frac{n^{1+\frac{1}{d}}}{x}$ ♦

These properties include the fact that our geometric expanders of dimension 3 will have $O\left(n^{\frac{5}{3}}\right)$ edges. Although the known lower bound would appear to imply that these results cannot be improved upon by using a lower dimension geometric expander, it is good to see that using dimension 2 geometric expanders does not work well empirically:

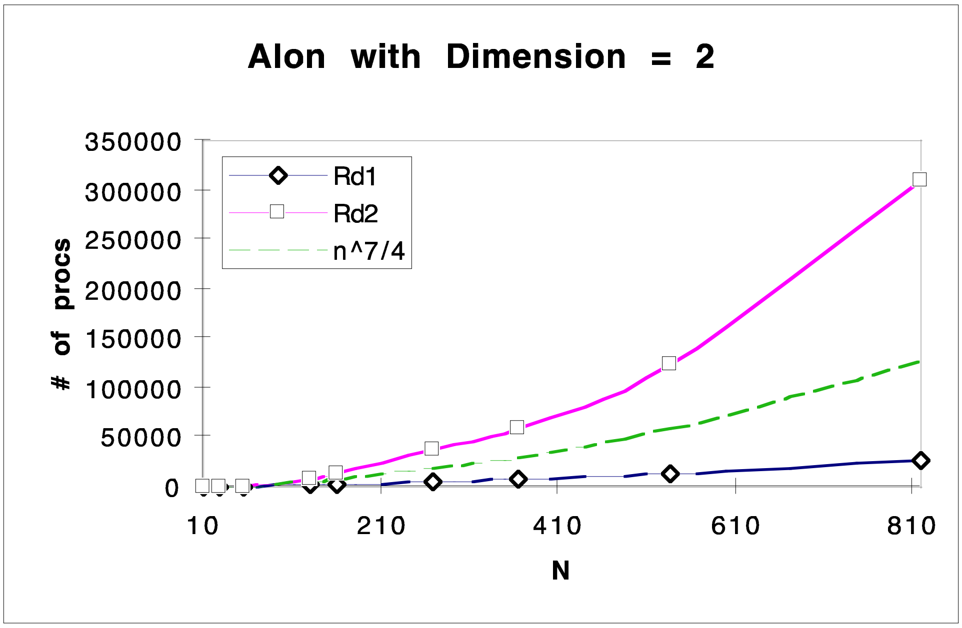

Figure 3.12

Section 3.4: Theorizing how various algorithms would impact [HH2]

In [HH2] a constructive algorithm for sorting in k rounds was presented for k>2. It is not applicable in the k=2 case since it builds upon merging using 2 rounds. The algorithm is recursive in nature, sorting sublists using some number of rounds before merging using the remaining rounds. The non-constructive result of $\mathrm{Sort}(n,2)=O\left(n^{\frac{5}{3}}(\log n)^{\frac{1}{3}}\right)$ from [BT1] would improve the results in [HH1].

We can easily discuss the improvements if the dimension 3 algorithm is $O\left(n^{\frac{5}{3}}\right)$ and Sort(n,2) turns out to be $\Theta\left(n^{\frac{5}{3}}\right)$. In the case of Sort($n$,4) [HH2] had us sort sublists in 1 round and then merge these lists in 3 rounds using $O\left(n^{\frac{20}{13}}\right)$ processors. If instead we were able to sort sublists in 2 rounds using this $\Theta\left(n^{\frac{5}{3}}\right)$ processors and then merge the sublists in 2 rounds, we could accomplish our task using $O\left(n^{\frac{3}{2}}\right)$ processors. Similar improvements could be made for all other values of k>3 using this algorithm.

Additionally, in the k=4 case where the previous algorithm did not begin to give good results until n was at least 8192 this new version begins to give good results with smaller values of n (as small as 931). In Figure 3.14 and Table 3.15, when "HH w/ Golub" and HHg are referred to, it is showing the number of processors used if the dimension 3 geometric expanders are used to sort in 2 rounds. These results are used only to help visualize the differences that an algorithm which used around $n^{\frac{5}{3}}$ processors could make.

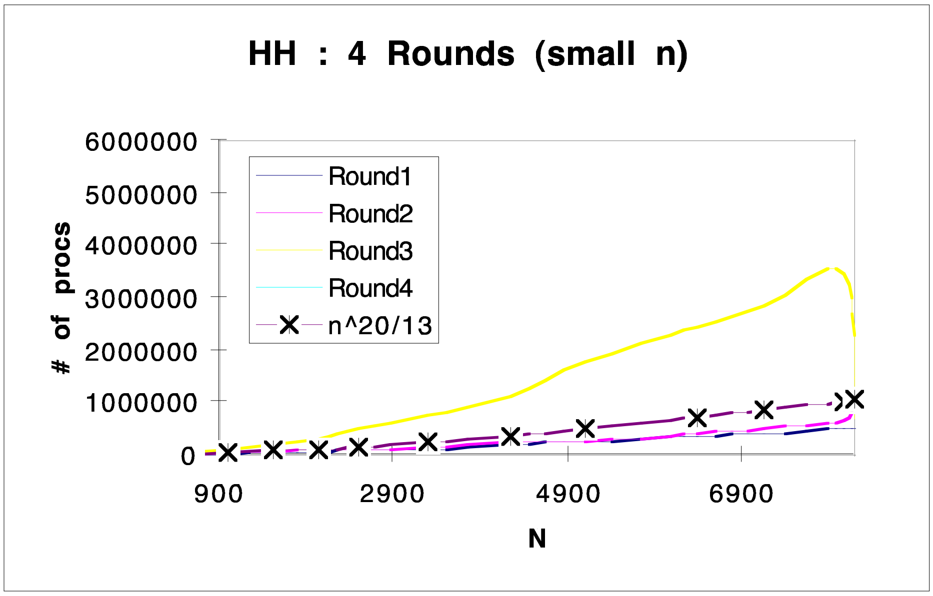

Figure 3.13

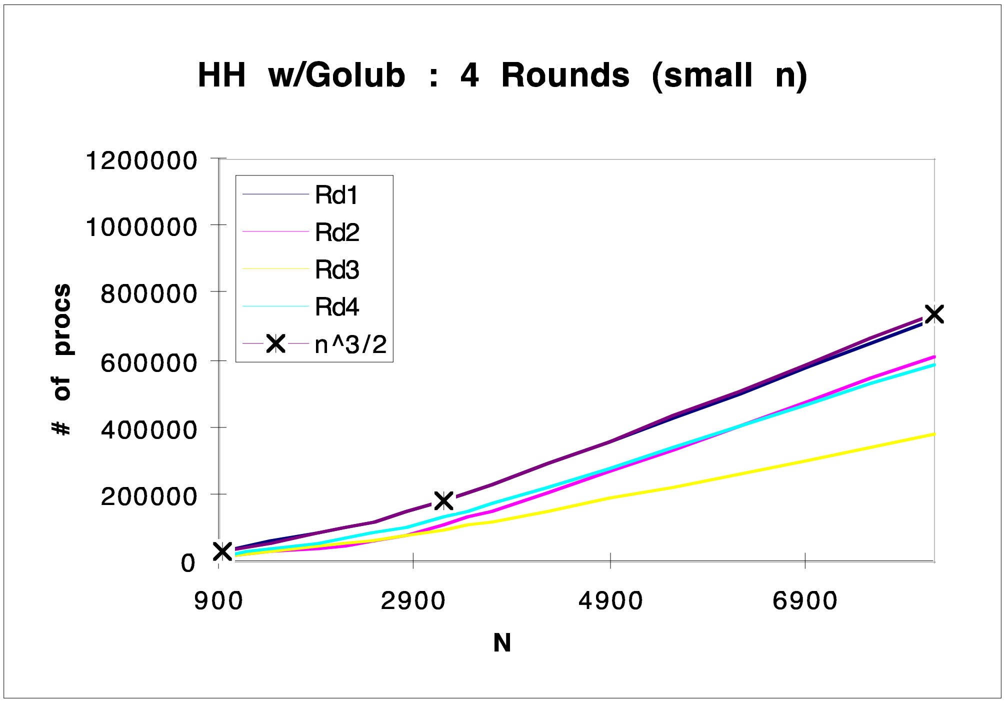

Figure 3.14

While the limitations that blocksize presented in [HH2] regarding choices of values of n which would work (if the actual n was smaller, we would pad upwards to the next largest n which would divide well into blocks) still exist, they are no more an issue here than before.

The following is a chart showing the number of processors required to sort in the specified number of rounds using the original [HH2] algorithm (HH), that algorithm with Alon's $O\left(n^{\frac{7}{4}}\right)$ result (HHa) and that algorithm with a $O\left(n^{\frac{5}{3}}\right)$ result (HHg). This last case is given again for the purpose of visualizing the potential impact of the result. [Note: If a given algorithm has the value f in the below chart, that means it will require $O\left(n^{f}\right)$ processors.]

Rounds	HH	HHa	HHg	
3	8/5 (1.60000)	8/5 (1.60000)	8/5 (1.60000)	
4	20/13 (1.53846)	26/17 (1.52941)	3/2 (1.50000)	
5	28/19 (1.47368)	22/15 (1.46666)	23/16 (1.43750)	
6	24/17 (1.41176)	24/17 (1.41176)	24/17 (1.41176)	
7	176/127 (1.38583)	76/55 (1.38182)	26/19 (1.36842)	
8	80/59 (1.35593)	188/139 (1.35252)	194/145 (1.33793)	
9	200/151 (1.32450)	200/151 (1.32450)	62/47 (1.31915)	*
10	64/49 (1.30612)	64/49 (1.30612)	212/163 (1.30061)	*
11	464/359 (1.29248)	40/31 (1.29032)	1556/1213 (1.28277)	
12	1592/1249 (1.27462)	488/383 (1.27415)	100/79 (1.26582)	
13	512/407 (1.25799)	512/407 (1.25799)	1108/883 (1.25481)	*
14	1136/911 (1.24698)	1136/911 (1.24698)	536/431 (1.24362)	*
15	1168/943 (1.23860)	704/569 (1.23726)	3896/3161 (1.23252)	

Table 3.15

The rows that have a * next to them represent cases where HHg would give an improvement over HH when HHa does not.

This can also be represented by the graph:

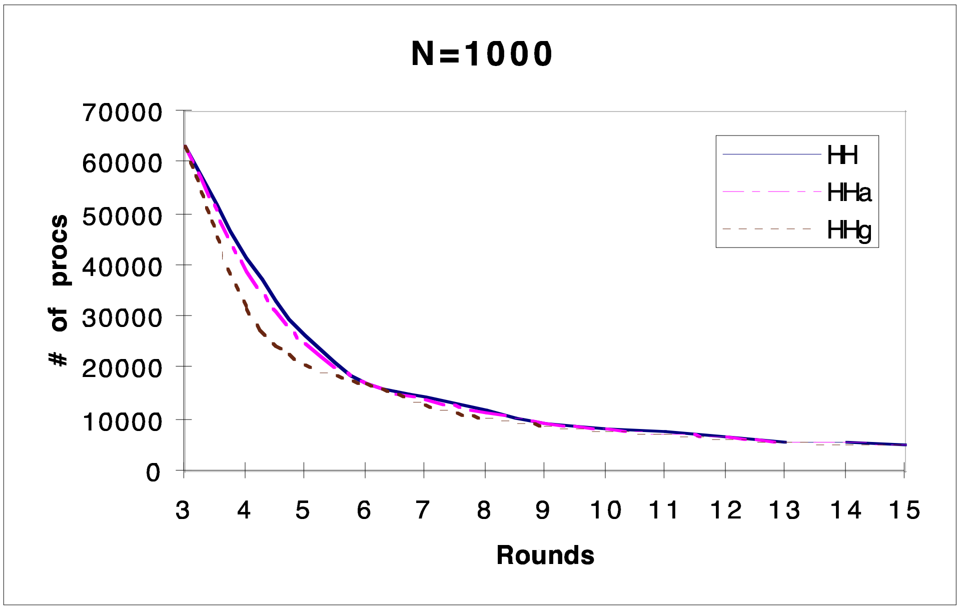

Figure 3.16

In many cases, HHa performs slightly better than HH. In all of these cases and several more, HHg would perform slightly better than HH or HHa. We have a non-constructive algorithm [Pip1] which performs better and will later discuss a constructive algorithm [WZ1] which theoretically performs. However, it should be noted that by using either (HHa) or (possibly HHg) whenever the recursive algorithm gets to a place where it needs to sort in 2 rounds only direct implications (rather than full transitive closure) will be computed during that part of the algorithm. This will give us the associated benefits of requiring only direct implications at some levels of these sorts.

Section 3.5: Summary of empirical results

In this case, my empirical results led to the possibility of a new constructive upper bound for this problem. Although the performance of the 2 round sorting algorithm using dimension 3 geometric expanders has not yet been formally described, the empirical results imply that there is something of interest happening in this case.

I can again address one of the philosophies that I presented earlier:

<u>Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.</u>

By implementing Alon's algorithm, I was able to find that the number of actual comparisons remaining in the second round appear to grow at a slower rate than the proof suggested. Since a flexible implementation existed, I was able to experiment with factors such as the dimension of the geometric expander and observe the results. These observations have led me towards attempting to formally describe and prove these results. Additionally, the algorithm can be utilized with [HH2] in practice to improve upon the apparent performance of the algorithm.

Chapter 4 : Alon, Azar and Vishkin's K-Round Sorting Algorithm

Section 4.1: Sketch of algorithm

Alon, Azar and Vishkin [AAV1] present a constructive proof for the following upper bound:

$$\text{Sort}(n,k) \text{ has expected value } O\left(n^{1+\frac{1}{k}}\right)$$

This comes from a randomized algorithm in which the exact comparisons are determined at run time using randomization. The expected number of required processors is then calculated for the algorithm.

Theorem 4.1[AAV1]: Sorting can be done in k rounds with an expected $O\left(n^{1+\frac{1}{k}}\right)$ processors

In the first round of this algorithm, $n^{\frac{1}{k}} - 1$ values $(t_1, t_2, \ldots, t_{n^{1/k}-1})$ are chosen at random and then compared to all n-1 other values requiring $O\left(n^{1+\frac{1}{k}}\right)$ processors. Between rounds, the n values are partitioned into $n^{\frac{1}{k}}$ blocks

$(A_1, A_2, \ldots, A_{n^{1/k}})$ based on the now ordered list of $n^{\frac{1}{k}} - 1$ values such that if $i<j$ then all members of A_i are less than members of A_j.

In the remaining k-1 rounds, each A_i is sorted. It is shown in [AAV1] that the expected number of processors required to do this will be $O\left(n^{1+\frac{1}{k}}\right)$. ♦

It should be noted at this point that this tells us that the expected number of processors is $O\left(n^{1+\frac{1}{k}}\right)$. In the event that there are more unanswered questions than there are available processors in a terminating round, that round will need to be split into multiple rounds. This will be referred to again in Section 4.3 below.

Section 4.2: Implementation of algorithm

This algorithm is a straight forward one to implement but does hide some coordination overhead that will be discussed later in this section. The following figures show some results of sorting 100 groups of values of the specified sizes and how many processors were required on average.

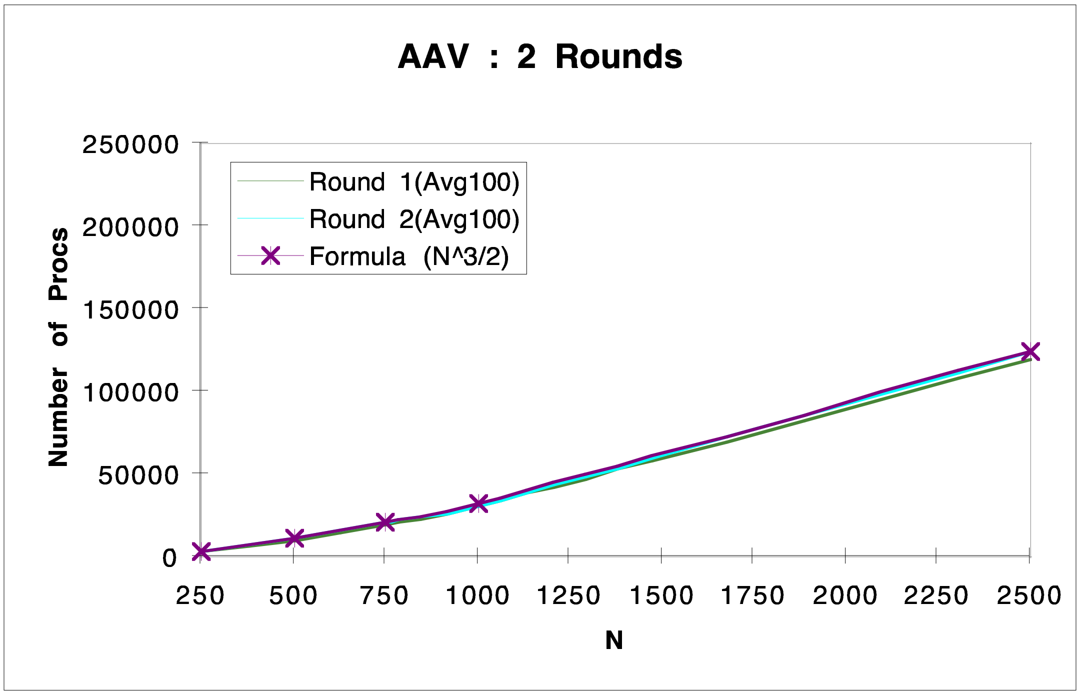

Figure 4.2

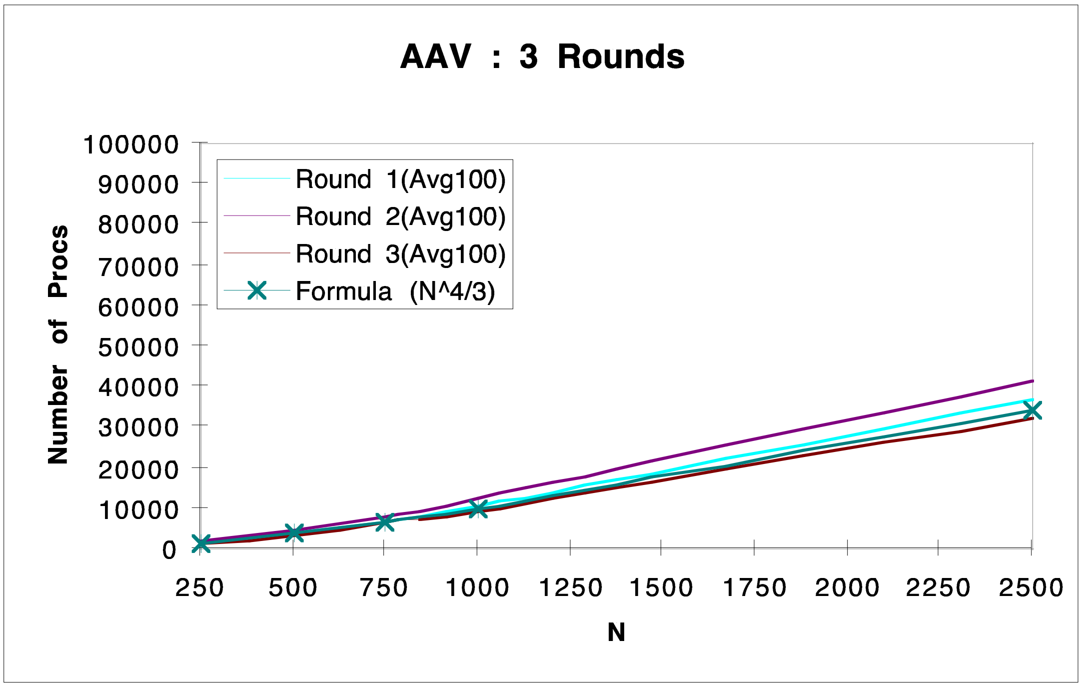

Figure 4.3

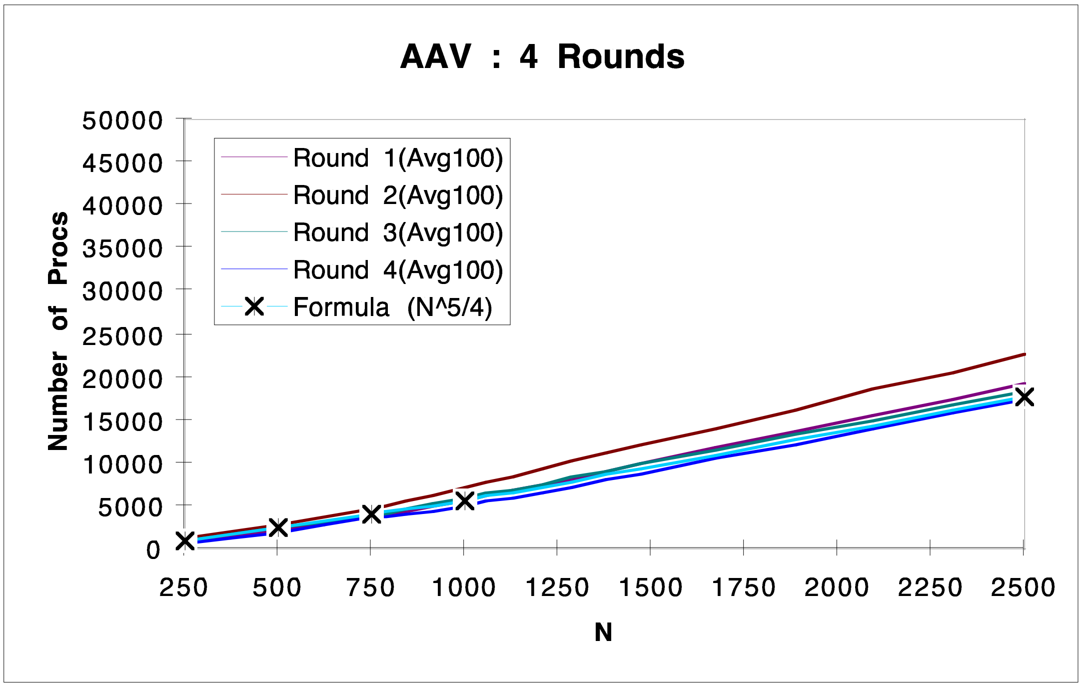

Figure 4.4

As predicted by the proof of this algorithm, the average number of processors required grows at the rate of $O\left(n^{1+\frac{1}{k}}\right)$.

While randomly choosing a group of $n^{\frac{1}{k}} - 1$ elements to use can be done quickly, there is (as there is with all of the algorithms we have looked at) work to be done between rounds. After Round 1 is completed, we need to create the $(A_1, A_2, \ldots, A_{n^{1/k}})$ partitioning of the values. All of the required comparisons have been done, but that information needs to be converted into actual partitionings. This requires that (a) the $(t_1, t_2, \ldots, t_{n^{1/k}-1})$ values be ordered and that each of the remaining $n - n^{\frac{1}{k}}$ values be placed into their appropriate cell.

Section 4.3: Thoughts on expected number of rounds

A question that arose during these experiments was that of how the expected number of processors would translate into an actual number of rounds. Since an underestimate of the number of processors required would lead to additional rounds being needed it would be in our best interest to observe the behavior of the algorithm to assist us in the selection of a number of processors to use.

The following figures show not only the average number of processors required in each round but the maximum number as well. Additionally, a line has been added representing twice the expected number of processors for improved visualization.

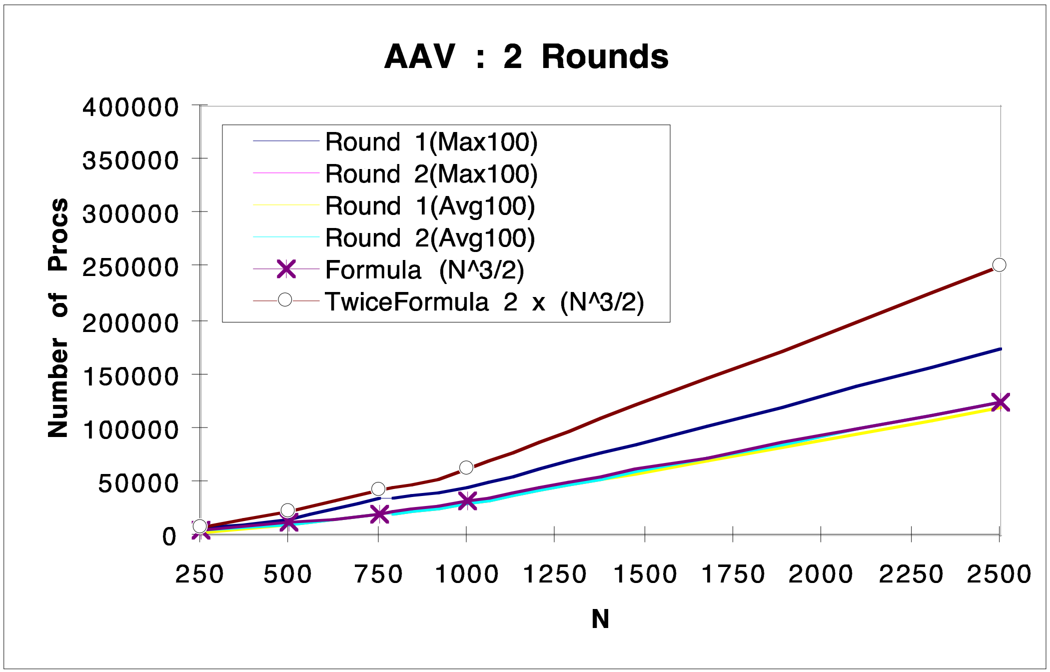

Figure 4.5

In the two round case, the average number of processors used in rounds one and two as well as the maximum number of processors used in round 2 are so close that they all overlap with the line representing the formula.

As in the two round case, many of the results being shown in the three and four round cases are so close that they overlap and appear as a single line or as a cluster of lines. It would be difficult to draw the graph as to show each line distinctly in grey scale colors.

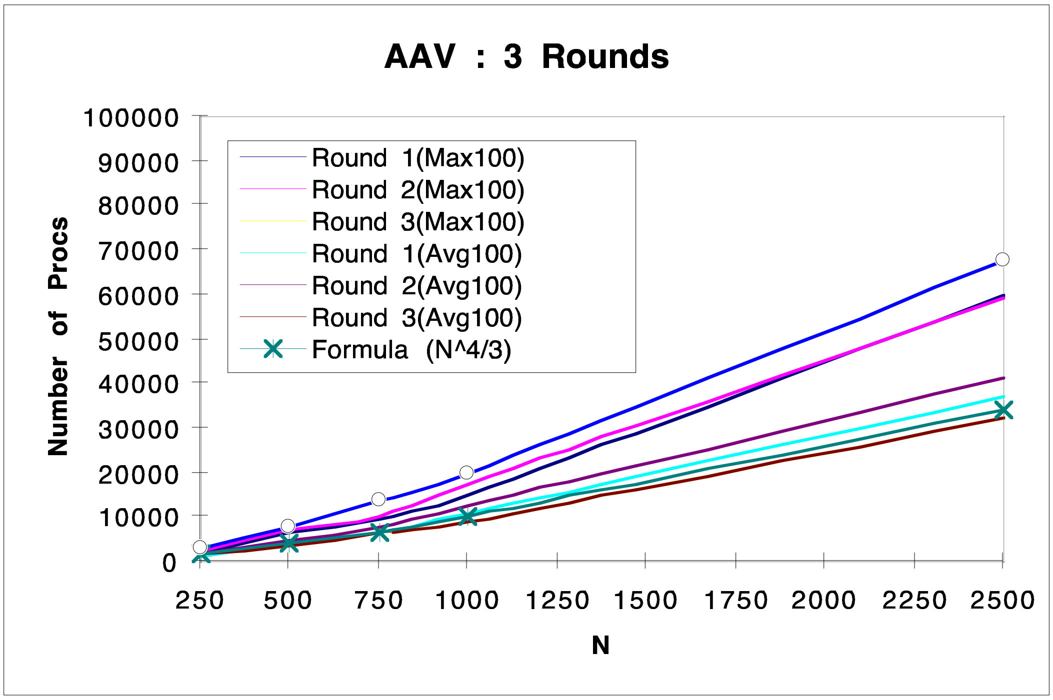

Figure 4.6

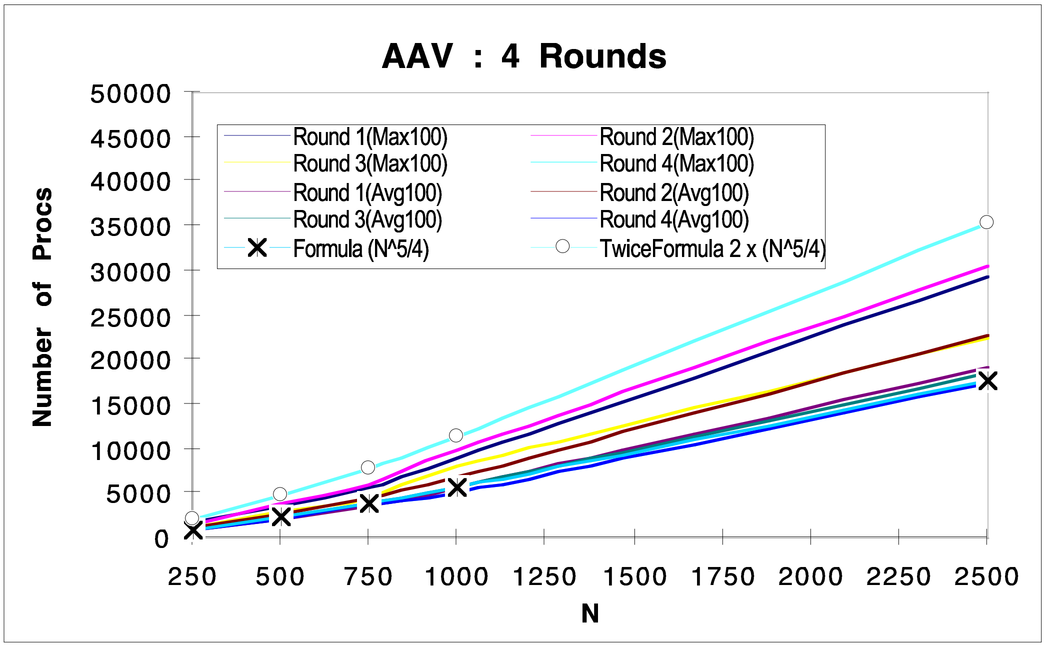

Figure 4.7

These empirical results give us reason to believe that if we allocate twice the expected number of processors we have a good chance of being able to avoid the requirement that additional rounds be used in the sort.

Section 4.4: Summary of empirical evidence

Through these experiments I can address both of the philosophies that I presented earlier:

<u>Philosophy 1(modified): Proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.</u>

The expected number of processors required by this randomized algorithm is lower than the existing constructive as well as non-constructive upper bounds. The above results show that this algorithm can be implemented with no more difficulties than other algorithms. It is also less than the lower bound for deterministic sorting, but that is due to the fact that this is the expected number of processors rather than a guaranteed number of required processors.

<u>Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.</u>

The above results show that the algorithm presented in [AAV1] behaves as expected. Additionally, it appears that by allocating twice the expected number of required processors, we will be able to sort in the desired number of rounds as well. While the proof does present us with an expected value, it does not address whether this average is obtain through radical swings above and below the average. The empirical results show that the answer to this question is “no”. The empirical results begin to build a better picture of the run-time behavior of the algorithm.

Chapter 5 : Bollobás and Thomason's K-Round Sorting Algorithm

Section 5.1: Sketch of algorithm

Bollobás and Thomason [BT1] present a constructive proof for the following upper bound:

$$\mathrm{Sort}(n,k) = O\left(n^{\frac{3}{2}+\frac{1}{2(2^{\frac{k+1}{2}}-1)}}\right) \quad \text{[for } k \text{ odd]}$$

Only direct implications (see Definition 3.1) are computed between rounds. Bollobás and Thomason comment that this algorithm is easy and will work for all values (specifically small values - eg: less than 1000) of *n*.

<u>Theorem 5.1: Sorting can be done in *k* rounds (where *k* is odd) using only direct implications with</u> $O\left(n^{\frac{3}{2}+\frac{1}{2(2^{\frac{k+1}{2}}-1)}}\right)$ <u>processors</u>

This algorithm is similar in some ways to [HH2] in that you first partition the original list into sublists, then sort those sublists and finally merge values back into a single ordered list. However, unlike [HH2] which took *j* rounds to sort the sublists and then used *k-j* rounds to execute all possible pairwise merges of those sublists, [BT1] uses k-2 rounds to recursively sort the sublists and then only 2 rounds to accomplish the merging of the sublists. Additionally, rather than computing the full transitive closure of the relationships learned, only direct implications need be computed.

The original list of n values is partitioned into m sublists (each of size n/m) where m is defined as $n^{1/(2^{\frac{k+1}{2}}-1)}$. These sublists are then ordered in k-2 rounds (using this algorithm recursively until k=1 in which case the traditionally method of comparing all values to all other values is used). Once the sublists are ordered, they are then recombined using the remaining 2 rounds.

In the first of the 2 remaining rounds comparisons are done in a similar manner to how you would search an ordered list in parallel:

$v \in$ original list of values

Take one of the ordered sublists $X = \left\{x_1, x_2, \ldots, x_{n/m}\right\}, v \notin X$

Compare v with $x_{\sqrt{n/m}}, x_{2\sqrt{n/m}}, \ldots, x_{n/m}$

(Note : This should be seen as creating sub-blocks of each sublist)

These comparisons will require $nm\sqrt{\frac{n}{m}}$ processors.

In the second of the 2 remaining rounds all remaining questions typically must be answered. However, after the previous round of questions, for every value *v* in the original list and every sublist it will be known which sub-block that value belongs to. It will be sufficient to compare the element to all elements in those sub-blocks (one per sublist) and then compute direct implications. Since there are *m* sublists each having a sub-block of size $\sqrt{\frac{n}{m}}$ for each of the original *n* values these comparisons will require $nm\sqrt{\frac{n}{m}}$ processors as well.

Since $nm\sqrt{\frac{n}{m}} = n^{\frac{3}{2}+\frac{1}{2(2^{\frac{k+1}{2}}-1)}}$ we have our result. ♦

Section 5.2: Implementation of algorithm

The algorithm is mostly straightforward to implement. The only tricky part is dealing with uneven block sizes and some floor/ceiling considerations when computing, for example, $\sqrt{\frac{n}{m}}$ for use in the program. The following figures show the results of using this algorithm in the 3 round, 5 round and 7 round cases.

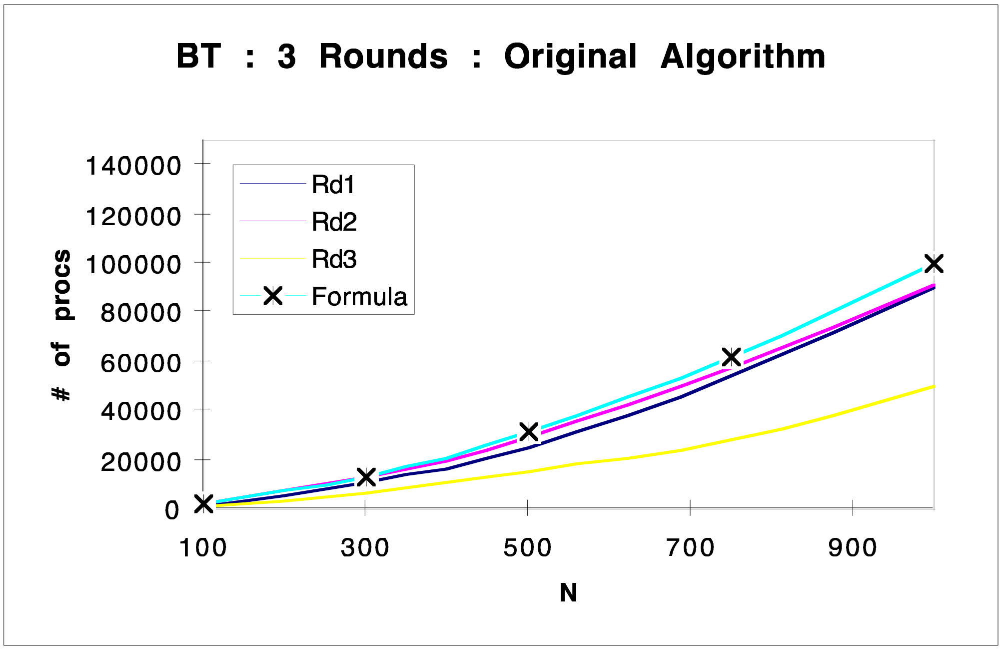

Figure 5.2

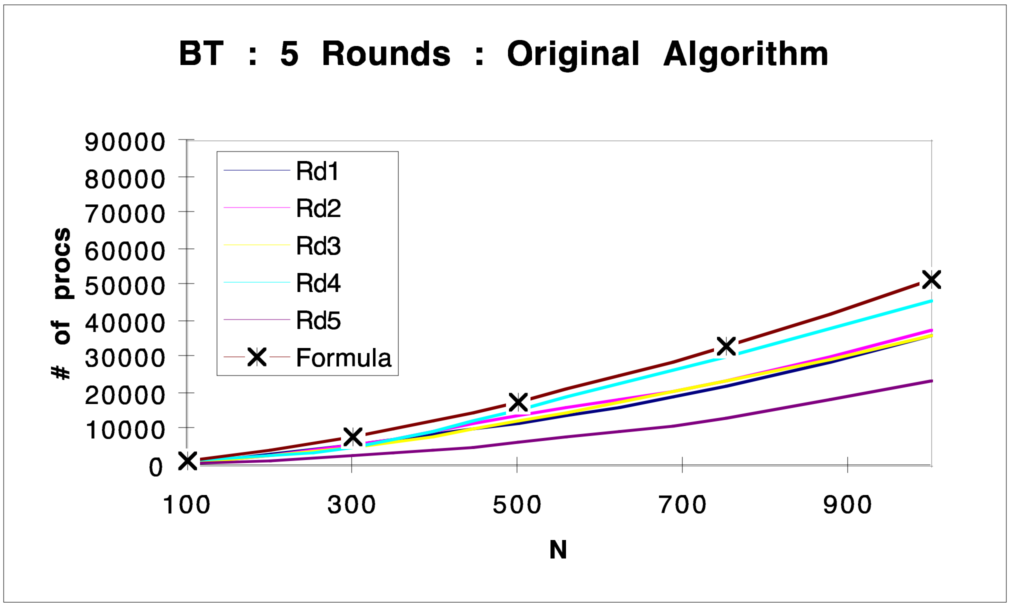

Figure 5.3

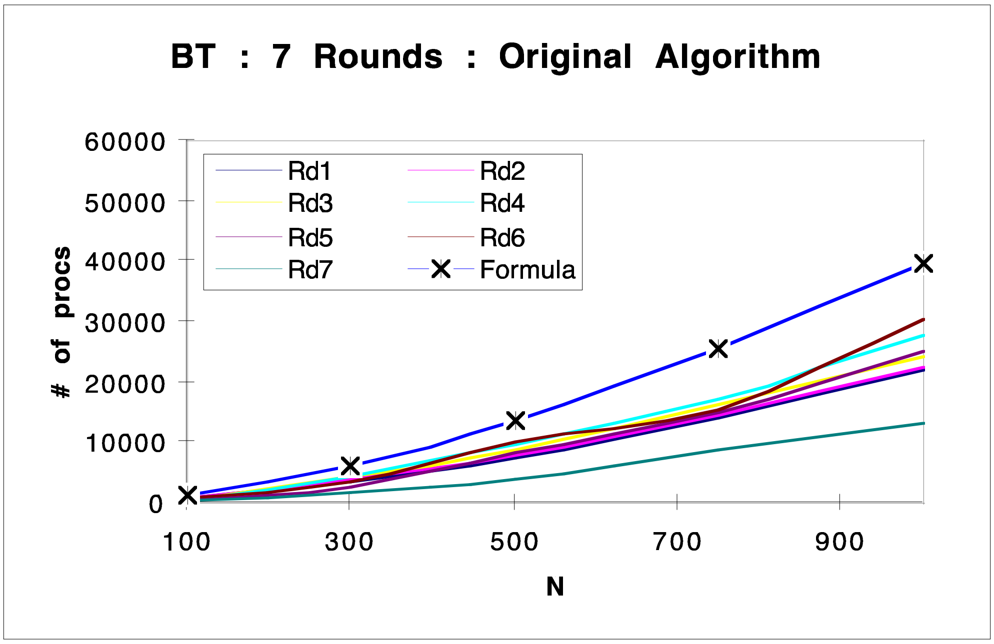

Figure 5.4

In all of these cases, the algorithm acted as expected, with each rounds' processor requirements having the same growth rate. Some minor oscillations occur in one of the rounds in the 7 round case, but these are due to the blocksizes and boundary cases as the sub-blocks become smaller and smaller in the later rounds. We witnessed similar oscillations in [HH2].

Section 5.3: Experimenting with "second" round

After implementing the algorithm as presented in [BT1] I became curious as the whether more information was being accumulated than was being used. Specifically, after the first "merging" round, it seemed that there would potentially be enough information to eliminate the need for some comparisons.

In my modified implementation, rather than waiting until after both of the "merging" rounds to do the computation of direct implications, I instead did these computations after the first round and then asked only remaining questions during the second round. I felt that since every value in the original list could be used as a direct intermediary between many pairs computing these direct implications could reduce the number of actual comparisons needed in the second merging round. The following graphs show that this does appear to be the case:

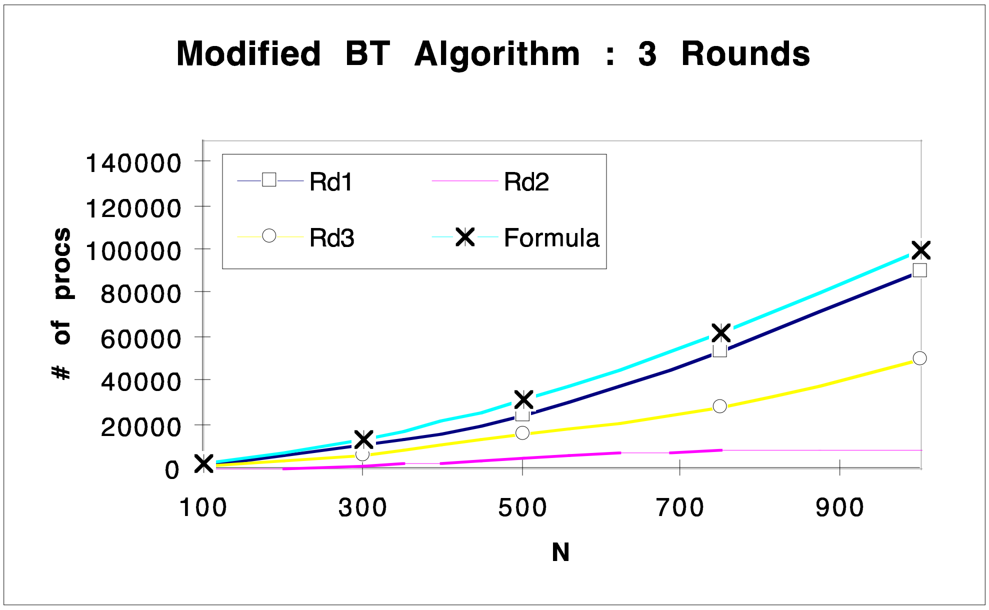

Figure 5.5

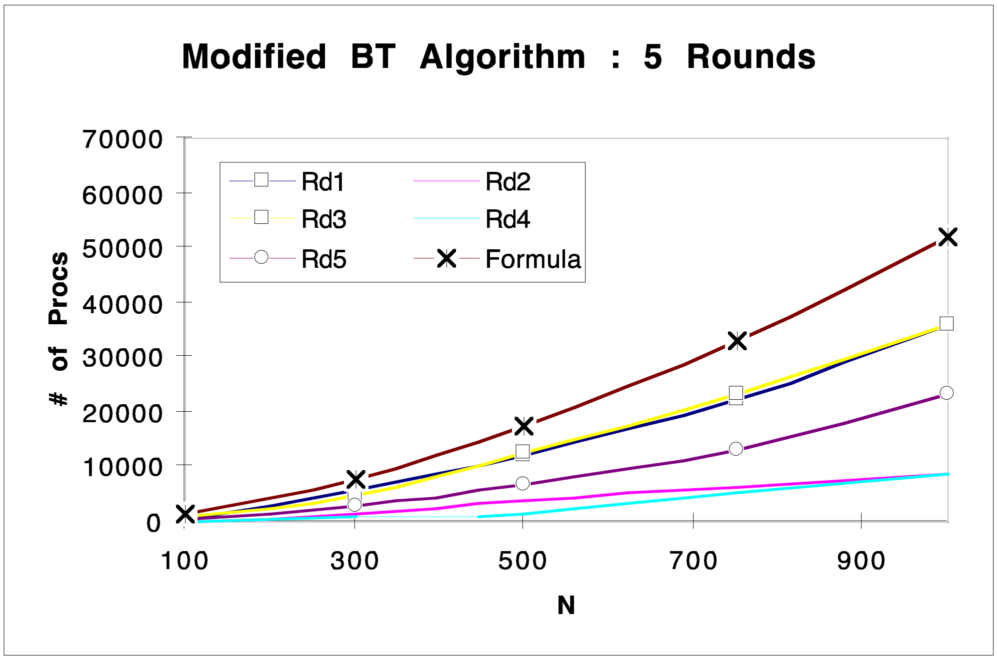

Figure 5.6

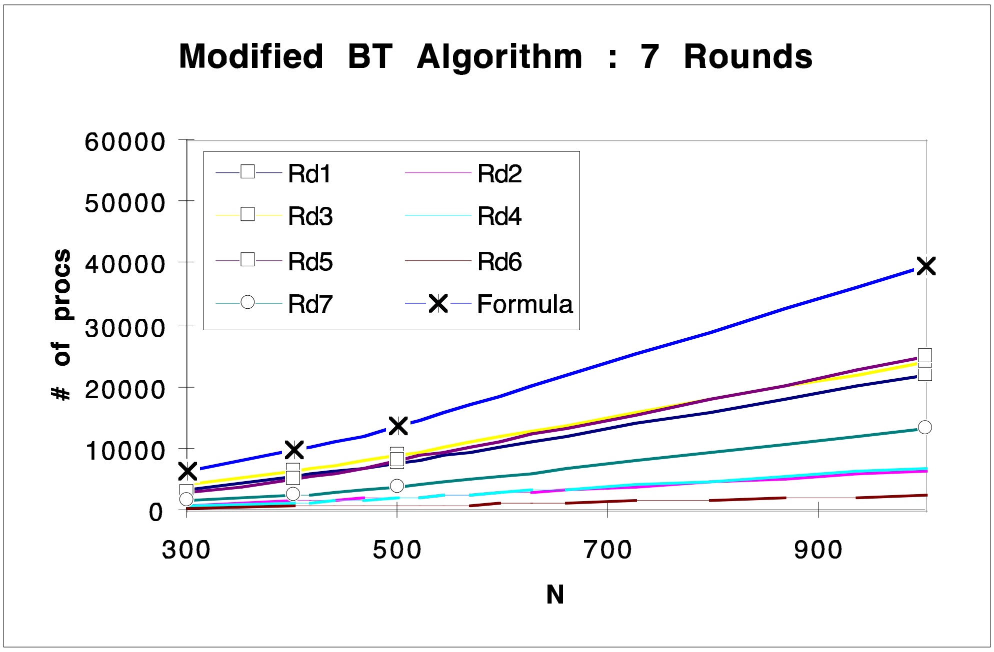

Figure 5.7

We can see from these that the number of processors that would be sufficient in the rounds that represent the “second” round of the “merging” is far fewer than required by the paper.

Unfortunately, it does not appear that we can take advantage of this by altering the number of sublists upon which we operate. While doing this can bring the processors used in the odd numbered rounds closer together (as shown below for the 3 round case), three things should be noted: (1) the odd numbered rounds all appear to be growing at the same rate, (2) most of the odd numbered round are already close and it is only the final round which is really lower and (3) there is no real effect towards bringing these closer to the growth rate of the number of processors used in the even numbered rounds.

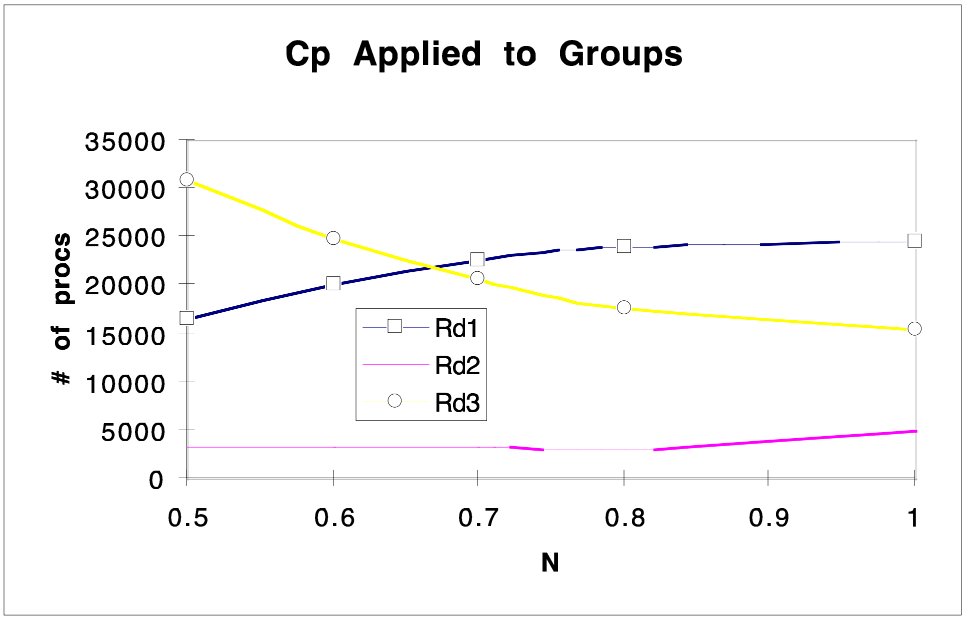

Figure 5.8

Another possibility would be to increase the size of the sub-blocks within each sublist. However, this in itself would have no impact on the number of processors required in the final round. The following shows (in the 3 round case) how the first two rounds are affected by altering the size of the sub-blocks:

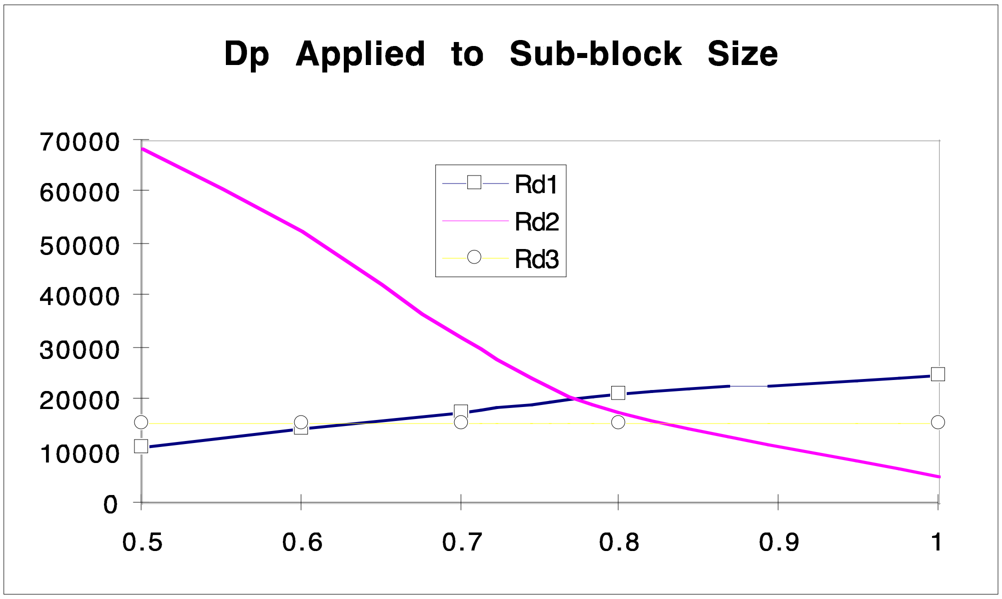

Figure 5.9

We do find that choosing a slightly different sub-block size could slightly decrease the number of processors required in the first round before the number of processors required in the second round increased too much. However, the growth rate still appears to match that of the original formula. These factors may, however, be of interest when working on a particular architecture.

Section 5.4: Comparing HH and BT algorithms

While this algorithm does give a slightly worse upper bound than [HH1] did, it does hold two possible advantages. The first of these advantages is the fact that it requires only direct implications rather than full transitive closure. The second is that it will work on smaller length inputs (*n*) than [HH1] will and does not have the strict limitations for the choice of *n* that the blocksize issues enforce in [HH1].

As we've been observing algorithms through empirical studies, it might be interesting to observe the relative performance of these two algorithms for some "small" lists. While the [BT1] algorithm does require more processors, the relative number of processors required in these trials did not exceed a factor of 2.6 for values of *n* between 500 and 10,000.

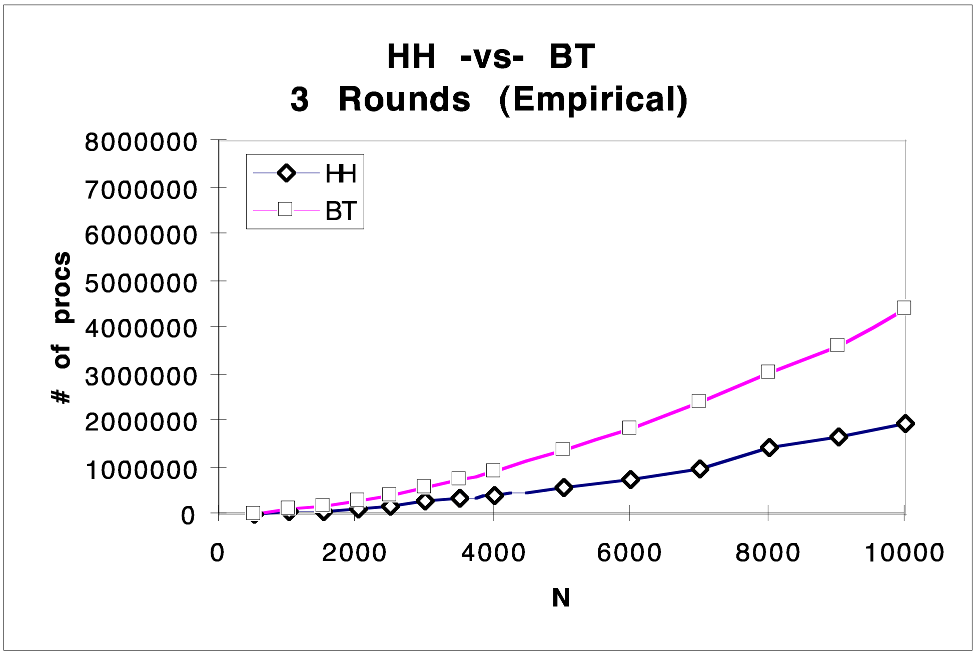

Figure 5.9

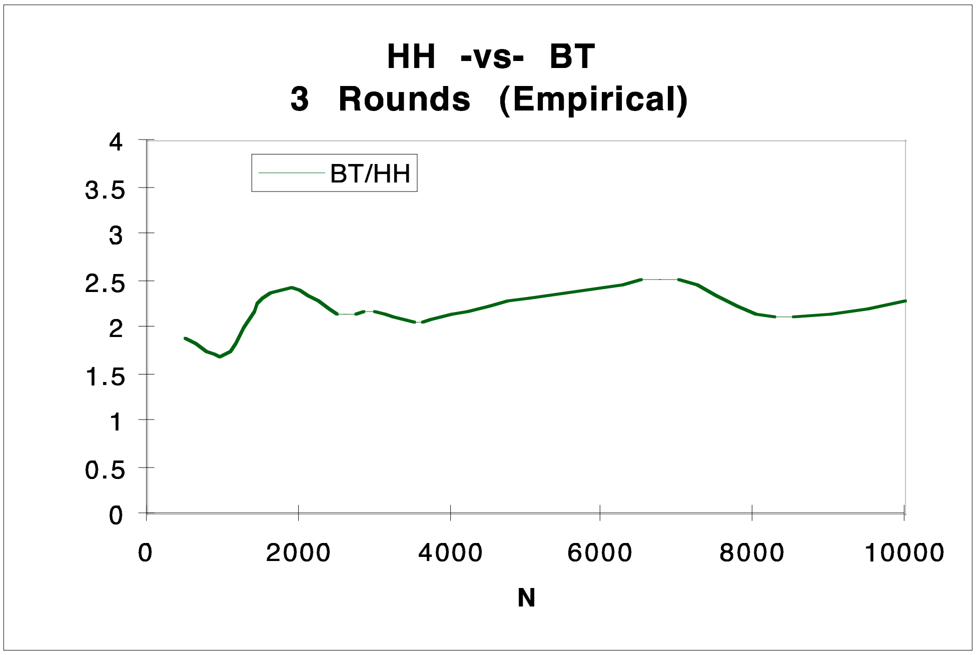

Figure 5.10

Section 5.5: Summary of empirical evidence

Through these experiments I address one of the philosophies that I presented earlier:

<u>Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.</u>

By implementing Bollobás and Thomason's algorithm, I was easily able to begin to test my theory that there was more information gathered during the first round of merging than was actually used by the paper's algorithm. Additionally, once having preliminary confirmation of this I was easily able to begin to test two sub-theories which said that this additional information could not actually be used to any significant benefit. I would note that interesting results are not always helpful ones.

Chapter 6 : Wigderson and Zuckerman's K-Round Sorting Algorithm

Section 6.1: Sketch of algorithm

Wigderson and Zuckerman [WZ1] present a constructive proof of the following upper bound:

$$\mathrm{Sort}(n,k) = \mathrm{O}\left(n^{1+\frac{1}{k}+o(1)}\right)$$

Their algorithm is based upon Pippenger's non-constructive sorting algorithm [Pip1]. Pippenger's algorithm presents a non-constructive proof of the existence of a category of expanding graphs which can be used well in sorting problems. Wigderson and Zuckerman present a constructive proof of the existence of a category of expanding graphs similar to the ones used by Pippenger. They then go on to show how these graphs can be used in various applications, including sorting.

Wigderson and Zuckerman construct their expanders using a function known as an extractor. Extractors are functions which essentially take a long string of bits with a short string of random bits and return a string of nearly random bits. An

$(n,m,t,\delta,\varepsilon)$-extractor takes the length of the "long" input string as *n*, the "short" input string as *t* and the output string as *m*. Additionally, the "nearly random" quality of the output is described in terms of δ and ε. They then use these extractors to build expanders.

Given an $(n,m,t,\delta,\varepsilon)$-extractor $E:\{0,1\}^n \times \{0,1\}^t \rightarrow \{0,1\}^m$ and an input of length $N = 2^n$ you can create a bipartite graph *H* using the extractor. We can describe this graph as $H: V \times W, V = \{0.1\}^n, W = \{0,1\}^m$ where you connect $x \in \{0,1\}^n$ to $z \in \{0,1\}^m$ iff $\exists y \in \{0,1\}^t$ such that $E(x,y) = z$. Once this graph has been built, any vertices in *W* with degree greater than $\frac{2N2^t}{M}$ (or more than twice the average number of nodes in *W*) are removed. We would then use this graph to build a new graph (*G*) which would then be used to sort. An edge (*x*,*z*) in the graph (which would represent a comparison between the x^{th} and z^{th} input values) is placed into graph *G* iff $x \in V, y \in W, z \in V, (x,y) \in H, (z,y) \in H$. The properties of an $(n,m,t,\delta,\varepsilon)$-extractor lead to *G* being an n^{δ}-expander.

Section 6.2: Requirements on size of input

In this algorithm, we need to consider the log of the size of the input when determining for which input lengths the sort will work. In Section 5.4 of [NZ1] a series of lengths for the {0,1}-tuples is given. It starts with l_0 as the largest integer such that $\sum_{i=1}^{\infty} \frac{l_0}{\left(1+\frac{\delta'}{4}\right)^i} \le \frac{\delta n}{4}$ and that $l_i = \frac{l_{i-1}}{1+\frac{\delta'}{4}}$. Notice that each l_i represents the length of a tuple, and that the sequence is non-increasing. If l_0 is not at least 1, then the extractor cannot be built. However, since we can solve the above summation to obtain that l_0 is the largest integer such that $l_0 \le \frac{\delta\delta' n}{16}$ we know that n must be at least 16 for l_0 to be at least 1. This would mean that our smallest input would need to be 2^{16} or 65,536.

Additionally, by writing a program to iterate through combinations of n, δ, ε and c, I found that n would need to be significantly higher than 16. In my experiments, the first such value of n which satisfied the requirements was 33. The results of this experiment are in the table which follows.

n/c	7.4	7.5	7.6	7.7	7.8	7.9	8.0	8.1	8.2
30	NB	NB	NB	NB	NB	NB	NB	DG	DG
31	NB	NB	NB	NB	NB	NB	NB	DG	DG
32	NB	NB	NB	NB	NB	NB	NB	DG	DG
33	NB	NB	NB	NB	L0	L0	L0	DG	DG
34	NB	NB	L0	L0	L0	L0	L0	DG	DG
35	L0	L0	L0	L0	L0	L0	L0	DG	DG

NB (*n* below value) = l_0 is zero (because $n < \frac{16}{\delta\delta'}$)

L0 = l_0 is non-zero with δ=0.5

DG (delta greater than 1) = δ' is greater than 1

Figure 6.1

Using the value 33 for *n* would give us an input size of 2^{33}, or over 8 billion. Our experiments to date have been on inputs ranging in sizes from as small as 100 to at most 15,000. This was partially due to time considerations in our simulations. Since we are only using a single processor during these simulations our ability to experiment on larger inputs is impaired. However, if the smallest inputs for which this technique would work are on the order of 8 billion, it is possible that this algorithm, while an interesting application of constructed expanders, would not be of use in many situations. It should be noted that Pippenger's probabilistic generation of graphs for sorting works well even on small inputs.

Section 6.3: Summary of empirical evidence

Through these trials and analysis I can address both of the philosophies that I presented earlier:

<u>Philosophy 1: Non-constructive proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.</u>

While Pippenger's algorithm [Pip1] is non-constructive, in my opinion it has two advantages over the above algorithm. First, Pippenger's works for much smaller values of n. Second, the implementation of Pippenger's algorithm is more straightforward. This second consideration is non-trivial. While an excellent mathematician will be able to understand the algorithm's design and proof of correctness and an excellent programmer will be able to implement the algorithm (once understood) on a specific hardware platform, it would likely require a most excellent team with both qualities to accomplish the job; one or the other would not suffice.

Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.

In this instance, we did not execute our simulations due to the required size of the input. However, it was our interest in the implementation and experimentation with the algorithm that led us to the point where we more closely analyzed the algorithm's requirements. Additionally, while a purely mathematical analysis delivered the $n \geq 16$ result, a computer program was used to help identify the probable starting point of $n=34$.

Conclusions

When beginning this research path, there were two major philosophies that I wished to demonstrate:

- Philosophy 1: Non-constructive proofs built upon probabilistic techniques can be just as good as, if not better than, constructive ones.

- Philosophy 2: Implementing algorithms (in simulated situations) can reveal new and interesting information as well as suggest new questions.

In the preceding chapters I have described my experiences with the works included in six different papers in the area of parallel sorting. The following chart summarizes which experiences helped affirm each of these two philosophies and how:

	Philosophy 1	**Philosophy 2**
Pippenger	Showed that this non-constructive proof can be used as the basis for the design of a	The value of p was fine tuned in a way that was not predicted by Pippenger's proof.

computer program that would in fact sort a given set of values in a given number of rounds using the stated number of processors. Once a suitable graph has been generated, it can be reused. The generation of a usable graph with the given factor of p was accomplished quickly.	Additionally, the performance of the graphs as the constant factor was changed implies that there is a sharp breakpoint before which graphs do not work well and after which they do.

Figure C.1

Häggkvist and Hell	While this proof was a constructive one, the issues that arose during implementation were more complex than with Pippenger's. Due to the pairwise merging at every level and associated requirement that all lists being merged have the same length, there is a good deal of extra overhead required to coordinate this.	In addition to the overhead for coordinating the blocks at each level, these empirical studies led to the observation that *n* needs to be sufficiently large so that there are no empty blocks at the bottom of the recursion. In the case of 4 rounds, this value is 2^{13} or 8192. For 5 rounds it is expected to be 2^{19}.

Figure C.2

Alon	The implementation of this algorithm was of the same level as Pippenger's algorithm. There was slightly more mathematical background necessary. It's use of partial transitive closure is it's most noteworthy distinction.	By implementing Alon's algorithm, I was able to find that the number of actual comparisons remaining in the second round appear to grow at a slower rate than the proof suggested. Since my implementation was flexible, I was able to experiment with factors such as the dimension of the geometric expander and observe the results. These observations have led me towards attempting to formally describe and prove these results.

Figure C.3

Alon, Azar and Vishkin	The expected number of processors required by this randomized algorithm is lower than the existing constructive as well as non-constructive upper bounds. The results showed that this algorithm can be implemented easily.	It appeared that by allocating twice the expected number of required processors, we would be able to sort in the desired number of rounds. While the proof presented an expected value, it did not address whether this average was obtained through radical swings above and below the average. The empirical results begin to build a better picture of the run-time behavior of the algorithm and show that they do not appear to suffer from radical swings.

Figure C.4

Bollobás and Thomason	The implementation of this algorithm requires slightly more planning than Pippenger's but is otherwise straight forward to implement.	By implementing Bollobás and Thomason's algorithm, we were able to test the theory that there was more information gathered during the first round of merging than was actually used by the paper's algorithm but that it would not be easy to capitalize on this information.

Figure C.5

Wigderson and Zuckerman	Although this was a constructive proof, it was found that Pippenger's algorithm has two potential advantages over it. First, Pippenger's works for much smaller values of n. Second, the implementation of Pippenger's algorithm is more straightforward.	In this instance, we did not execute our simulations due to the required size of the input. However, it was our interest in the implementation and experimentation with the algorithm that led us to the point where we more closely analyzed the algorithm's requirements.

Figure C.6

It is my belief that both of these philosophies are sound and that there is much to be gained from the empirical study of theoretical works in computer science.

This work is mean to be a beginning rather than the end. Some interesting questions which now exist include:

- Are the graphs being built and working well in the simulation of Pippenger's algorithm actually a-expanders?

- Could a-expanders with properties (relative to sorting) similar to Ramanujan graphs be generated probabilistically and tested more easily than the current a-expanders?
- Will Sort(n,2,2) be found to be $\Theta\left(n^{\frac{5}{3}}\right)$?
- Can any of these sorting algorithms be used towards selection in rounds?

Additionally, although the application studied here (parallel sorting in rounds) may not become practical for years if ever, there are other areas which the techniques and philosophies discussed can be applied. Some examples of these are:

- Probabilistically generated memory layouts.
- Probabilistically generated selection, merging and selection networks.

In chosing other problems to study, we would likely target ones in which the probabilities are close to 1 as in the case of Pippenger's algorithm.

References

[AA1] Alon, N., Azar, Y. (1988) Sorting, Approximate Sorting, and Searching in Rounds. *SIAM Journal on Discrete Mathematics*; 1:269-280.

[AAV1] Alon, N., Azar, Y., Vishkin, U. (1986) Tight Bounds for Parallel Comparison Sorting. *IEEE Symposium on Foundations of Computer Science*; 27:502-510.

[AKS1] Ajtai, M., Komlos, J., Szemeredi,E. (1983) Sorting in c log n Parallel Steps. *Combinatorica*; 3:1-19.

[AKS2] Ajtai, M., Komlos, J., Szemeredi,E. (1983) An O(n log n) Sorting Network. *ACM Symposium on Theory of Computing*; 15:1-9.

[AKSS1] Ajtai,M., Komlos,J., Steiger,W., Szemeredi,E. (1989) Optimal Parallel Selection Has Complexity O(log log n). *Journal of Computer and System Sciences*; 38(1):125-133.

[Alon1] Alon, N. (1986) Eigenvalues, Geometric Expanders, Sorting in Rounds, and Ramsey Theorey. *Combinatorica*; 6(3):207-219.

[BH1] Bollobas, B., Hell, P. (1985) Sorting and Graphs. *Graphs and Orders*;169-184.

[Bol1] Bollobas, B. (1988) Sorting in Rounds. *Discrete Mathematics*; 72:21-28.

[BT1] Bollobas, B., Thompson,A. (1983) Parallel Sorting. *Discrete Applied Mathematics*; 6:1-11.

[BT2] Bollobas, B., Thompson,A. (1986) Threshold Functions. *Combinatorica*; 7(1):35-38.

[Cole1] Cole, R. (1988) Parallel Merge Sort. *SIAM Journal of Computing*; 17(4):770-785.

DZ1 Dor, D., Zwick, U. (1995) Selecting the Median. *ACM-SIAM Symposium on Discrete Algorithms*; 6:28-37.

[DZ2] Dor, D., Zwick, U. (1996) Finding the alpha n-th Largest Element. *Combinatorica*; 16:41-58.

[DZ3] Dor, D., Zwick, U. (1996) Median Selection Requires (2 + epsilon)n Comparisons. *IEEE Symposium on Foundations of Computer Science*; 37:125-134.

[HH1] Haggkvist, R., Hell, P. (1981) Parallel Sorting with Constant Time For Comparisons. *SIAM Journal of Computing*; 10(3):465-472.

[HH2] Haggkvist, R., Hell, P. (1982) Sorting and Merging In Rounds. *SIAM Journal on Algebraic and Discrete Methods*; 3(4):465-473.

[Knuth1] Knuth, D. (1973) Sorting and Searching. Addison Wesley Publishing Co., Reading, MA.

[Krus1] Kruskal, C. (1983) Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*; C-32(10):942-946.

[Krus2] Kruskal, C. (1999) An exact bound for parallel merging. *Working paper.*

[LPS1] Lubotzky, A., Phillips, R., Sarnak, P. (1988) Ramanujan Graphs. *Combinatorica*; 8(3):261-277.

[NZ1] Nisan, N., Zuckerman, D. (1996) Randomness is Linear in Space. Journal of Computer and System Sciences; 52:43-52.

[Pip1] Pippenger, N. (1987) Sorting and Selecting in Rounds. *SIAM Journal of Computing*; 16(6):1032-1038.

[Val1] Valiant, L. (1975) Parallelism in comparison problems. *SIAM Journal of Computing*; 4(3):348-355.

[Vish1] Vishkin, U. (1987) An optimal parallel algorithm for selection. Advances in Computing Research, JAI Press Inc., Greenwich, CT.

[Will1] Williams, J. (1964) Algorithm 232 - Heapsort. *Communications of the ACM*; 7:347-348.

[WZ1] Wigderson, A., Zuckerman, D. (1993) Expanders that Beat the

Eigenvalue Bound: Explicit Construction and Applications. *ACM Symposium on Theory of Computing*; 25:245-251.