Scalability of Parallel Sorting on Mesh Multicomputers*

V. Singh,[‡] V. Kumar,[¶] G. Agha,[§] and C. Tomlinson[‡]

[‡]MCC
3500 West Balcones Center Drive
Austin, TX 78759

[¶]Computer Science Department
University of Minnesota
Minneapolis, MN 55455

[§]Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

This paper presents two new parallel algorithms QSP1 and QSP2 based on sequential quicksort for sorting data on a mesh multicomputer, and analyzes their scalability using the isoefficiency metric. We show that QSP2 matches the lower bound on the isoefficiency function for mesh multicomputers. The isoefficiency of QSP1 is also fairly close to optimal. Lang et al. and Schnorr et al. have developed parallel sorting algorithms for the mesh architecture that have either optimal (Schnorr) or close to optimal (Lang) run-time complexity for the one-element-per-processor case. Both QSP1 and QSP2 have worse performance than these algorithms for the one-element-per-processor case. But QSP1 and QSP2 have better scalability than the scaled-down variants of these algorithms (for the case in which there are more elements than processors). As a result, our new parallel formulations are better than these scaled-down variants in terms of speedup w.r.t the best sequential algorithms. We also present a different variant of Lang's sort which is asymptotically as scalable as QSP2 (for the multiple-element-per-processor case). We briefly discuss another metric called "resource consumption metric". According to this metric, both QSP1 and QSP2 are strictly superior to Lang's sort and its variations.

*Kumar's work was partially supported by Army Research Office grant # 28408-MA-SDI to the University of Minnesota and by the Army High Performance Computing Research Center at the University of Minnesota. Agha's work has been supported in part by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

1 Introduction

In this paper, we investigate the problem of parallel sorting on a two-dimensional mesh multicomputer architecture. We characterize the scalability of various algorithms using the formally-defined isoefficiency metric [15]. As described below, this metric helps assess the performance of a given parallel algorithm on a given architecture under realistic situations.

Let us first consider the disadvantages of using a metric other than isoefficiency. Many parallel algorithms for sorting are analyzed using the metric of run-time complexity when the number of processors equals the number of data elements (e.g., see [18, 23, 17]). However, this metric is not very useful for realistic situations when the number of data elements can far outnumber the number of processors. For example, one could easily expect to sort a million data elements, but it would be quite unreasonable to expect to have a million processor system to sort these elements.

When there are fewer processors (p) than there are data elements (N), one possibility is to *scale down* the parallel algorithm by making each physical processor emulate $\frac{N}{p}$ processors. If context-switching costs are

ignored, such a technique will result in a slow down of at most a factor of $\frac{N}{p}$ over the original parallel algorithm. If the original parallel algorithm is cost-optimal (i.e., if the processor-time product [2] of the original parallel algorithm is the same as the sequential time complexity of the best sequential algorithm), then this technique works well; the resulting scaled-down parallel algorithm still has the same processor-time product. Thus, one could choose the parallel algorithm with the best run-time complexity, and scale it down appropriately to derive the best parallel algorithm for the case in which fewer processors are available.

Unfortunately, if the original parallel algorithm is not cost-optimal, then scaling down can result in an inferior algorithm. It is entirely possible that a scaled-down variant of an algorithm with worse run-time complexity for the one-element-per-processor case (or, rather, a minor variation of such an algorithm which handles multiple elements per processor) might grossly outperform scaled-down variants of other algorithms with superior one-element-per-processor run-time complexity. It should be observed that cost-optimal parallel sorting algorithms (for the one-element-per-processor case) are available only for practically infeasible architectures such as one of the variations of PRAM. Thus, the above run-time complexity metric is of little use in assessing the performance of parallel sorting algorithms in realistic situations. We need a metric that helps us compare the performance of different parallel algorithms for different combinations of p and N (or some other indicator of problem size). The above run-time metric is inadequate, as it only deals with a 1-D curve on the 2-D space (of p and problem size).

The *isoefficiency metric* initially introduced in [15] is one metric that can be used to compare the performance of parallel algorithms when both problem size and number of processors may vary. The isoefficiency of a parallel algorithm reflects its scalability; i.e., its ability to effectively utilize increasing number of processors. The speedup obtained by a parallel algorithm is usually dependent upon the hardware architecture (such as interconnection network, the CPU speed, speed of the communication channel) as well as certain characteristics of the parallel algorithm (such as the degree of concurrency, and overheads due to communication, synchronization, and redundant work). Due to these overheads, the speedup will saturate at a certain limit for an architecture and a problem instance of a fixed size. For many parallel algorithms, a larger problem (e.g., a sorting problem with a larger list of data elements) will have a higher speedup limit. The isoefficiency metric relates the problem size to the number of processors necessary for linear speedup. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel algorithms [15, 22, 16, 8, 14]. An important feature of isoefficiency analysis is that it succinctly captures the behavior of a parallel algorithm in relation to the given parallel architecture.

This paper presents two new parallel algorithms QSP1 and QSP2 based on sequential quicksort for sorting data on a mesh multicomputer, and analyzes their scalability. We show that QSP2 matches the lower bound on the isoefficiency function for mesh multicomputers. The isoefficiency of QSP1 is also fairly close to optimal. Lang et al. [18] and Schnorr et al. [23] have developed parallel sorting algorithms for the mesh architecture that have either optimal [23] or close to optimal [18] run-time complexity for the one-element-per-processor case. Both QSP1 and QSP2 have worse performance than these algorithms for the one-element-per-processor case. But QSP1 and QSP2 have better scalability than the scaled-down variants of these algorithms (for the case in which there are more elements than processors). As a result, our new parallel formulations are better than these scaled-down variants in terms of speedup w.r.t the best sequential algorithms. We also present a different variant of Lang's sort which is asymptotically as scalable as QSP2 (for the multiple-element-per-processor case). We briefly discuss another metric called "resource consumption metric". According to this metric, both QSP1 and QSP2 are strictly superior to Lang's sort and its variations.

2 Definitions and Assumptions

This paper considers parallel sorting of data elements that are initially distributed with uniform density over a mesh-connected multicomputer. The output data is also distributed over the multicomputer and sorted in row-major order (or some other well-defined order) across processors; data is sorted within processors as usual.

We introduce some terminology used in the rest of the paper. We consider a parallel processor consisting of an ensemble of p processing units each of which – for purposes of determining the complexity – runs at the same speed.

When a parallel algorithm is run on multiple processors, time spent by an individual processor P_i can be split into t_e^i (e for essential), the time spent in useful computation; and t_o^i (o for overhead), the time spent in doing communication, idling and other work which

would not have been performed by the best sequential algorithm but is necessitated due to parallel processing. The execution time on p processors, T_p, satisfies $T_p = t_e^i + t_o^i$.

The *computation size* T_e of a problem is defined to be the amount of computation (in units of time) taken by an optimal sequential algorithm. Clearly, $T_e = \sum_{i=0}^{p-1} t_e^i$. We define $T_o = \sum_{i=0}^{p-1} t_o^i$. Clearly, we have $T_o + T_e = p \times T_p$.

Note that earlier papers on scalability by the authors have used the term *problem size* in place of *computation size*. The reason for the switch in the terms is that problem size is often used to refer to input size.

The *speedup* S of an algorithm on p processors is defined to be the ratio $\frac{T_e}{T_p}$. The *efficiency* is defined as follows:

$$E = \frac{S}{p} = \frac{T_e}{T_p \times p} = \frac{T_e}{T_e + T_o}$$

We assume that the time to deliver a message of size m from a processor to its neighbor's processor is $(k_s + k_b m)$, where k_s is the start-up time and k_b is inverse of the channel bandwidth. Note that for current multicomputers, the start-up time is an order of magnitude larger than k_b. Hence (to simplify some of the expressions in the analysis), we assume the time to deliver a message of size 1 from a processor to its neighbor's processor is k_s.

Finally, we let k_c be the time for a processor to compare two elements and k_d be the time for moving a data element within the processor's memory. Let $k_{QS} N \log N$[1] be the average-case time for a processor to internally sort N elements using quicksort, where k_{QS} is a constant and $k_{MS} N \log N$ is the worst-case time for a processor to internally sort N elements using mergesort, where k_{MS} is a constant. We will use quicksort as the base optimal sequential algorithm in this paper for computing speedups and efficiencies of parallel algorithms.[2] Let the total number of elements to be sorted on the multicomputer be N and the total number of processors be p (arranged in a $\sqrt{p} \times \sqrt{p}$ mesh). Let r be the number of elements per processor at the beginning of the sorting algorithm, which is $\frac{N}{p}$.

[1] Note that all logs in this paper are with base 2 except where specified otherwise.

[2] We consider only sequential algorithms for randomly distributed elements based on compare-exchange. Thus we ignore algorithms such as radix sorting.

3 Scalability of Parallel Algorithms

If a parallel algorithm is used to solve a given problem of a fixed computation size (i.e., T_e), then the efficiency decreases as the number of processors increases. This property is true of all parallel algorithms. For a large class of parallel algorithms (e.g., parallel DFS [15], parallel 0/1 knapsack [19], parallel connected components [12], and parallel shortest path algorithms [13]), the following additional property is also true:

- For any given number p of processors, the efficiency of the parallel algorithm goes up monotonically (i.e., it never goes down, and approaches a constant e, s.t. $0 < e \leq 1$) if it is used to solve problem instances of increasing size.

We call such algorithms *scalable* parallel algorithms. In these algorithms, efficiency can be maintained at a desired value (between 0 and e) for increasing number of processors provided that the computation size is also increased. Note that for different parallel architectures, the computation size may have to increase at different rates (w.r.t. the number of processors) in order to maintain efficiency. The rate of growth of the computation size, w.r.t. the number of processors, that is required to keep the efficiency fixed essentially determines the degree of scalability of these parallel algorithms (for a specific architecture). For example, if the computation size is required to grow exponentially w.r.t. the number of processors, then the algorithm-architecture combination has a poor scalability, as it would be difficult to obtain good speedups for a large number of processors (unless the computation size being solved is enormously large). On the other hand, if the computation size needs to grow only linearly w.r.t. the number of processors then the parallel algorithm is highly scalable; i.e., it can easily deliver linear speedup for arbitrarily large numbers of processors. Since all problems have a sequential component (taken to be at least one arithmetic operation in this paper), the computation size must asymptotically grow at least linearly w.r.t. the number of processors to maintain a given efficiency. If the computation size needs to grow as $f(p)$, where p is the number of processors, to maintain an efficiency E, then $f(p)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f(p)$ w.r.t. p is defined to be the *isoefficiency curve* for efficiency E. It is possible to have different isoefficiency functions for different efficiencies. In this paper, isoefficiency functions are the same for all values of E

within some range $0 < E \leq e \leq 1$. Therefore, for the sake of convenience in the rest of the paper, we will refer to isoefficiency functions and isoefficiency curves without reference to a specific efficiency.

4 A Naive Parallel Formulation of Quicksort

Quicksort [1] is a recursive algorithm that repeatedly partitions an unsorted list into two smaller sublists using one of the members of the original list as a *pivot*. One of the sublists contains elements less than or equal to the pivot and the other contains elements greater than the pivot. After recursively "quicksorting" each of the two unsorted sublists, the two sorted sublists can be concatenated in the appropriate order to produce the sorted list corresponding to the original unsorted list. Recall that the average time to quicksort a list of length N is $k_{QS} N \log N$, where k_{QS} is a constant.

The following naive parallel variant of quicksort [21] is frequently used to illustrate the utility of high-level parallel programming languages [9, 20].

In this algorithm, the entire unsorted list is stored initially in one processor. This processor partitions the list into two sublists. It hands one sublist to an idle processor and keeps the other sublist. After the two sublists are recursively sorted and the second processor has returned its sorted sublist, the first processor combines the two sorted sublists to produce the sorted list corresponding to the original unsorted list. Note that the recursive partitioning of the work continues until all processors are busy. Then each processor invokes the sequential mergesort to sort the sublist allotted to it.[3]

As we show in [25], this algorithm is highly unscalable. Even on an architecture on which inter-processor communication is free (e.g., PRAM) the isoefficiency of this algorithm is $\Theta(2^{kp} \times p)$ (here k is some constant). Informally, the reason for poor scalability of naive parallel quicksort is that it has a large sequential component. Initial partitioning, which is done sequentially, takes $O(N)$ time, whereas the whole sequential algorithm takes only $O(N \log N)$ time. Hence, irrespective of any other factors (such as sequential components in the recursive steps, communication overheads, etc.), the problem size must grow at least exponentially to mask the effect of the large sequential component.

The reader should note that there are other parallel formulations of quicksort on PRAM [2, 10, 5] which can be shown to have much better scalability.

[3]The reader may wonder as to why we don't use sequential quicksort at this point. The reason is as follows. When many processors compute a sequential sort in parallel, the time taken is the worst of all of them. Quicksort is not a good algorithm here because it is efficient only on the average. In the worst case, it takes time $\Theta(L^2)$ for a list of length L. As the number of processors increases, the time taken will approach $\Theta(L^2)$ if quicksort were used as the sequential sort.

5 More Scalable Formulations of Quicksort

Here we present a new parallel quicksort algorithm and some of its variations and show that all of them are more scalable on a mesh than the naive parallel quicksort (on a PRAM). All time complexity results in this section are for the average case. The following is an informal description of the basic algorithm QSP1 and its variations. An element is selected as the pivot. The number of elements bigger and smaller than the pivot are counted. Then the smaller elements are moved to the processors that come first in the row-major ordering (and larger elements are moved to the processors that come later in the row-major ordering). This step is repeated recursively until a partition fits only within one processor. At that time, a sequential mergesort is used for sorting the local elements.

5.1 Algorithm QSP1

Here we describe QSP1 in greater detail. First, we will describe the single-element-per-processor case and then extend it to the multiple-elements-per-processor case.

Let us look at the recursive partitioning step that is applied over and over again. An example is given in figure 1. The figure shows the distribution of elements before and after the partitioning step. If the original array of processors is of size $\sqrt{p} \times \sqrt{p}$, then the set of processors in every partitioning step will consist of zero or more rows of length $\sqrt{p}$ with possibly a partial row on top and possibly a partial row at the bottom.

The partitioning step consists of four phases—A, B, C, and D.

Phase A: Send the pivot (in the first processor of the set of processors ordered in row-major order) to all the other processors. This is done by using a binary tree threading of the processors as shown in figure 2. The pivot is transmitted from the root of the tree towards the leaves. Since the longest path from the root to the leaves has $2\sqrt{p}$ hops, this step takes $O(\sqrt{p})$ time. Phases B and C (described below) also use the

same binary tree threading of the processors and take $O(\sqrt{p})$ time.

Phase B: In this phase, information is gathered at each node and passed up the tree. In particular, each node collects the numbers of greaters (elements greater than the pivot) and lessers (elements smaller than or equal to the pivot) in both the left and right subtrees. Using the status of the element at the node itself, the proper information is propagated to the parent.

Phase C: In this phase, information is propagated down the tree to enable each element to move to the proper position in the two processor partitions. Assuming row-major order of the processors as before, the greaters (lessers) occupy the same relative positions in the greater (lesser) partition that they occupied in the pre-partitioned set of processors. Each processor receives from its parent the next empty positions in the greater and lesser partitions. Depending on the status of the element in the processor, the proper information for the two subtrees is then propagated onwards.

Phase D: In this phase, each element moves to the proper position in the greater or lesser partition.

The partitioning process recurses until partitions are of size 1. That completes the sorting of the entire array of processors in row-major order.

Figure 1: QSP1 Example With One Element Per Processor

In order to extend this algorithm to apply to the case of multiple elements per processor, the following changes have to be made in the algorithm:

Phase A: One of the elements in the first processor is chosen as the pivot. Pivot distribution remains the same. This step takes time $k_s \times 2\sqrt{p}$, since the worst case distance traveled in the vertical and horizontal directions is $\sqrt{p}$ hops each. The distance in the vertical direction will halve on the average in every iteration but we will ignore this effect.

Phase B: On receiving the pivot, each processor divides its elements into bags of lessers and greaters. Also, it maintains the number of lessers and greaters. Information propagated from the leaves to the root of the tree takes into account that number of lessers and greaters at each node can be greater than 1. Since there are $\frac{N}{p}$ elements per processor, the time for comparison is $\frac{N}{p}k_c$. Propagating the information back takes time $k_s \times 2\sqrt{p}$ (if the small k_b term is ignored).

Phase C: Similarly, the information propagated from the root to the leaves about the next free processor (in each partition) is modified to account for multiple elements per processor. In particular, the two partitions are separated at a processor boundary. Therefore, the number of elements per processor may differ somewhat in the two partitions. The next free

Figure 2: Binary Tree Threading

processor is specified along with the number of elements that can be added to the processor. This takes time $k_s \times 2\sqrt{p}$.

Phase D: Elements are moved in large messages to take advantage of the amortization of startup time k_s. Notice that a message from one processor may have to be split across two destination processors. This can be taken care of by having a processor send its message to any one of the two destination processors. Then, the portion for the other destination processor can be sent in one hop. We will ignore the second data movement over one hop since it is negligible compared to the first. Using Kunde's algorithm for permuting data across a mesh multicomputer [17], the time taken for a permutation is $3\sqrt{p}(k_s + k_b\frac{N}{p})$.[4] Since two messages leave each processor—one for the lesser partition and the second for the greater partition—we will double the start-up time k_s charged to each processor to account for it. Therefore, the total time is $3\sqrt{p}(2k_s + k_b\frac{N}{p})$.

Theorem 1 *The isoefficiency function of QSP1 is* $\Theta(2^{k\sqrt{p}\log p}\sqrt{p}\log p)$*, where k is a constant.*

The sum of the time spent on all the parts is

$$k_s \times 2\sqrt{p} + (\frac{N}{p}k_c + k_s \times 2\sqrt{p}) + k_s \times 2\sqrt{p} + 3\sqrt{p}(2k_s + k_b\frac{N}{p})$$

The recursive step is applied until each partition occupies a single processor. Since the partitioning step does not cut the number of processors in exactly half each time, it turns out that the total number of iterations is more than $\log p$. The average number of iterations is less than or equal to $3\log p + 6$ [7]. We approximate this to be $3\log p$.

Therefore, the total time spent on all the iterations is given by the following formula:

$$(k_s \times 2\sqrt{p} + \frac{N}{p}k_c + k_s \times 2\sqrt{p} + k_s \times 2\sqrt{p} + 3\sqrt{p}(2k_s + k_b\frac{N}{p}))3\log p \qquad (1)$$

Finally, each processor needs to mergesort its elements sequentially. If the number of elements per processor remained constant, this would require time $k_{MS}\frac{N}{p}\log\frac{N}{p}$. However, the number of elements per processor varies somewhat across processors because the partition boundaries are forced to be at processor boundaries. Let $I\frac{N}{p}$ be the maximum number of elements in a processor. We are unable to determine the average value of I analytically. ¿From Monte Carlo simulations, we found that I is less than a small constant up to very large values of p.[5] Therefore, the maximum time for sequential sorting will be $k_{MS}\frac{IN}{p}\log\frac{IN}{p}$, which is approximately $k_{MS}\frac{IN}{p}\log\frac{N}{p}$ (ignoring a small log term).

Note that formula 1 for the total time spent on all the iterations needs to be modified as well to account for the increased density. We assume the maximum density throughout the process of parallel sorting. (The expression thus obtained is an upper bound. The actual time should be less.) Therefore, the new formula is obtained by replacing N by IN in the previous formula. The new formula is:

$$(k_s \times 2\sqrt{p} + \frac{IN}{p}k_c + k_s \times 2\sqrt{p} + k_s \times 2\sqrt{p} + 3\sqrt{p}(2k_s + k_b\frac{IN}{p}))3\log p$$

On adding the time for the initial sorting across processors and the sequential sorting at the end, we get the total time T_p as given below.

$$\begin{aligned} T_p &= (k_s \times 2\sqrt{p} + \frac{IN}{p}k_c + k_s \times 2\sqrt{p} + k_s \times 2\sqrt{p} + \\ &\quad 3\sqrt{p}(2k_s + k_b\frac{IN}{p}))3\log p + Ik_{MS}\frac{N}{p}\log\frac{N}{p} \\ &= (6\sqrt{p}k_s + 3\sqrt{p}(2k_s + k_b\frac{IN}{p}) + \\ &\quad k_c\frac{IN}{p})3\log p + Ik_{MS}\frac{N}{p}\log\frac{N}{p} \end{aligned}$$

Therefore,

$$T_e + T_o = p \times T_p$$

$$= p^{1.5}(12k_s + 3Irk_b + \frac{Ir}{\sqrt{p}}k_c)3\log p + rpk_{MS}I\log r$$

If N is kept constant, and p increases, then E decreases because T_e stays the same while $T_e + T_o$ increases. If p is kept constant and N increases, then E increases because T_e increases faster than $T_e + T_o$. If p increases, then for constant E, N should grow such that $\frac{T_e}{T_e+T_o}$ stays constant. Since the largest term in the order of complexity in $(T_e + T_o)$ is $9Irk_bp^{1.5}r\log p$, for constant efficiency,

$$N\log N = \Theta(N\sqrt{p}\log p)$$

or

$$N = \Theta(2^{k\sqrt{p}\log p})$$

[4] $3\sqrt{p}$ may be reduced to $2\sqrt{p}$ if each processor is allowed to use more than a constant amount of memory.

[5] The second half of next subsection discusses a technique for eliminating I.

where k is a constant.

$$T_e = \Theta(N \log N) = \Theta(2^{k\sqrt{p}\log p}\sqrt{p}\log p)$$

This function for T_e is the isoefficiency function for QSP1. Note that this is fairly close to the lower bound $\Omega(2^{c\sqrt{p}}\sqrt{p})$ (where c is a constant) derived in [25].

□

5.2 Algorithm QSP2

This variant differs from QSP1 in that the partitioning is done alternately in the vertical and horizontal dimensions. The advantage of this is that the maximum distance (between a processor containing the pivot and any other processor) within each partition is reduced by a factor of two (on the average) after each set of one horizontal and one vertical partitioning. As a result, time taken by the steps A, B, C, and D in all but the first few iterations is quite small compared with the time taken by the first few iterations. Hence the overall complexity of steps A, B, C, and D for all iterations is $\Theta(N\sqrt{p})$ (as opposed to $\Theta(N\sqrt{p}\log p)$ for QSP1). As a result, we have the following theorem (proof in [25]):

Theorem 2 *The isoefficiency function of QSP2 is* $\Theta(2^{k\sqrt{p}}\sqrt{p})$*, where* k *is a constant.*

We note a small variation of QSP2 (and analogously of QSP1) which has similar order of time complexity but reduces run-time by a constant factor. In this variation, partition boundaries are permitted to fall within processors. This will lead to a reduction in maximum density compared to QSP2 and thereby reduce per processor calculation time. However, since a processor may belong to up to 4 partitions, its communication ports may be congested, thereby slowing communication by up to a factor of 4. The slowdown will probably be much less since current multicomputers can perform a lot of the message-passing communication tasks concurrently for all ports on each processor. Hence, this variant promises to improve QSP2 for large numbers of processors. However, we are currently unable to analytically quantify the exact extent of improvement.

6 Langsort and its variations

Lang et al. [18] presented a parallel sorting algorithm for SIMD mesh parallel computers for the case in which each processor has 1 element per processor. Due to space constraints, this section is very cryptic. The reader is referred to [25] for more details.

6.1 The Basic Langsort Algorithm

In Langsort, the basic operations allowed are *exchange* and *compare-exchange*. The exchange operation exchanges two elements in adjacent processors (one element in each processor). The compare-exchange operation exchanges the two elements if and only if the elements are not in correct sorted order (given an ordering of the two processors involved). Otherwise, the compare-exchange operation does nothing. We show in [25] that

$$T_p \leq (9\sqrt{p} - 9)(k_s + k_b + k_c) \tag{2}$$

6.2 Scaled-Down Variant of Langsort

If we scale down Langsort to the case where N elements need to be sorted using $p < N$ processors, the resulting algorithm will have complexity no more than

$$T_p \leq (9\sqrt{N} - 9)(k_s + k_b + k_c) \times \frac{N}{p} \tag{3}$$

Now, we make some further optimizations to the simple scaled-down variant (1) by aggregating smaller messages into longer ones and (2) by using an optimal sequential sorting algorithm right at the beginning within each processor. We call this new algorithm ELS1 and we show in [25] that

Theorem 3 *The isoefficiency function for algorithm ELS1 is undefined (i.e., ELS1 is not scalable).*

There is another algorithm designed by Schnorr and Shamir [23] for the single-element-per-processor case on a mesh multicomputer that is asymptotically faster by a constant factor than Langsort. Schnorr and Shamir's algorithm has the worst-case time-complexity of $3\sqrt{p}$ (plus some lower order terms). It has been shown that this is the lowest worst-case complexity one can obtain for mesh multicomputers [23]. Schnorr and Shamir's algorithm can be modified in a way similar to ELS1 and will suffer from the same lack of an isoefficiency function (more details in [25].

6.3 Algorithm ELS2

This section describes another variant of Langsort to deal with multiple elements per processor. In this variant, each *compare-exchange* in the original Langsort is replaced by a *sublist compare-exchange*. A sublist compare-exchange takes two sorted sublists and creates two sorted sublists. One of the output sublists contains the higher half of the elements from the input sublists and the second contains the lower half. We show in [25] that

Theorem 4 *The isoefficiency of algorithm ELS2 is* $\Theta(\sqrt{p}\,2^{c\sqrt{p}})$*, where c is a constant.*

7 Performance Predictions

Figures 3, 4, and 5 plot efficiency versus number of processors and number of data elements for QSP1, ELS1, and ELS2 respectively based on the theoretical analyses presented earlier. The portions of the plots where the number of processors is greater than the number of data elements should be ignored. The values given to the parameters were: $k_s = 10$, $k_b = k_c = k_d = 1$, $k_{MS} = 3$, $k_{QS} = 2$, $I = 5$.

Clearly, ELS1 is unscalable; efficiency drops as the number of elements increase for a fixed number of processors. QSP1 is better than ELS1 except for a small set of combinations of number of data elements and number of processors. For moderate to large problem sizes, ELS1 gives very poor efficiencies. We noted earlier in the paper that ELS2 and QSP1 have almost equal isoefficiency functions. However, the single-element-per-processor Langsort has higher efficiency compared to the single-element-per-processor QSP1. This should lead to better asymptotic efficiency for ELS2 (by at least a constant factor) if the computation sizes are increased at the rate specified by the optimal isoefficiency function. As shown in the plots, ELS2 seems to be better for all combinations of the number of data elements and the number of processors.

Figure 3: Efficiency Plot for QSP1

Note that complexity analyses in this paper ignore some small terms that may make an asymptotic difference of a small constant multiple in the derived time of computation. Theorems do not get affected by these small constant multiples because they deal with order of complexity. Performance predictions depend on constants and may be off by a small amount. However, the trends in the performance predictions will be correct.

Figure 4: Efficiency Plot for ELS1

8 The Resource Consumption Metric

So far we have essentially judged the parallel algorithms by the speedup obtained (w.r.t. to the best sequential algorithm). This method of evaluation is reasonable if the entire multicomputer is dedicated to a single problem, as all that matters is the time in which the problem is solved (on the given number of processors). Consider the case of two parallel algorithms P1 and P2. P1 takes more time to finish solv-

Figure 5: Efficiency Plot for ELS2

ing the problem than P2 on a given number of processors (and thus provides worse speedup than P2). But while executing P1 most of the processors remain idle, whereas P2 keeps all the processors busy all the time. Clearly, P1 can become more desirable than P2 if we time-share the parallel processor among a number of problems; i.e., if the entire p-processor system is used to solve more than one problem. (This is different than the case in which the parallel processor is partitioned into 2 subsystems, and each is used to solve a different problem.)

Let us define **resource consumption** of a parallel algorithm as the sum of all machine cycles consumed by the parallel processor. A parallel processor containing p processors has a total of pT cycles available during time T. The resource consumption of a parallel algorithm is the number of these cycles used by the algorithm, leaving the remaining ones to be used by other parallel algorithms that are time-sharing the parallel processor.

The total number of comparison steps performed by all the processors in the execution of QSP1 and QSP2 is clearly $\Theta(NlogN)$, as collectively they perform exactly the same operations that would be performed by the sequential quicksort. The reader can verify that the total time spent by the processors in communication as well as in bookkeeping operations for QSP1 and QSP2 is no more than $\Theta(NlogN)$. Hence, the resource consumption of QSP1 and QSP2 is $\Theta(NlogN)$. On the other hand, in ELS2, all the processors remain busy for a duration that is of the same order as the run-time of the algorithm. Hence for ELS2, the resource consumption is $\Theta(N\sqrt{p} + N\log N - N\log p)$. (For ELS1, the resource consumption is $\Theta(N^{1.5})$, which is even worse than that for ELS2.) Clearly, for $p > (\log N)^2$, QSP2 and QSP1 would consume fewer machine cycles than ELS2.

9 Conclusions

Isoefficiency analysis provides insights into the usefulness of various parallel sorting algorithms for mesh multicomputers. As discussed in Section 7, ELS1, the optimized scaled-down variant of Langsort, performs very poorly compared with QSP1 and QSP2 for most practical combinations of computation and architecture sizes. The number of data points in a typical sorting application should far exceed the number of processors. QSP1 and QSP2 are much better at benefiting from the increased problem size compared with ELS1. As discussed in Section 7, ELS2 is able to obtain better speedups than QSP1 and QSP2. However, QSP1 and QSP2 are superior to ELS2 in terms of overall resource consumption; i.e., the number of computing cycles consumed by QSP1 and QSP2 are smaller than that for ELS2 for many combinations of T_e (or N) and p. This becomes important when the parallel processor is being shared among many different applications.

It must be noted that ELS1 is a natural algorithm that one would derive by simply following the data-parallel paradigm in [11]. The main argument of the proponents of this paradigm is that the user can assume the existence of as many virtual processors as needed by the problem and the compiler, the run-time system, or algorithm designer can simply map more than one virtual processor to a single processor as necessary. From the analysis of ELS1, it is clear that this is not a good technique for at least some problems. Even the scaled-down variant of Schnorr and Shamir's optimal one-element-per-processor algorithm has no isoefficiency (i.e., is not scalable). The methodology of first uncovering all the inherent parallelism in the problem, and then mapping the concurrent activities to the parallel architectures has been promoted by a number of researchers [3, 24, 4]. This is one of the motivations behind developing the parallel algorithms that are in the NC class [6]. This is also similar to the idea advocated by Athas and Seitz in the context of the Actor paradigm.[6] Our results show that this methodology does not result in the best possible parallel algorithms at least for some problems. It is important to keep the scalability issues in focus while designing the algorithms.

Some researchers have claimed that mesh-based multicomputers with cut-through routing can emulate a fully-connected network. From our analysis, it is clear that for asymptotic order of time complexity of sorting, mesh with cut-through routing is no more powerful than a mesh with simple routing (i.e., store-and-forward routing). It has been shown elsewhere that the same is true for FFT [8] and some parallel algorithms for shortest path [16]. Of course, for many other parallel algorithms (including some for shortest path), mesh with cut-through routing has much better scalability than mesh with store-and-forward routing.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms. Computer*

[6] They believe that random mapping should work well in a lot of cases. In all cases, they feel that an algorithm-independent strategy will suffice.

Science and Information Processing, Addison-Wesley, 1983.

[2] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[3] Bill Athas. *Fine Grain Concurrent Computations*. PhD thesis, Computer Science Department, California Institute of Technology, 1987. Also published as technical report 5242:TR:87.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.

[5] Bogdan S. Chlebus and Imrich Vrto. Parallel quicksort. *Journal of Parallel and Distributed Processing*, 1991(to appear).

[6] Stephen A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathematique*, XXVIII:99–124, 1981.

[7] L. Devroye. A note on the height of binary search trees. *Journal of Association of Computing Machinery*, 33:489–498, 1986.

[8] Anshul Gupta and Vipin Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990. An extended version of the paper is available as a technical report from the Department of Computer Science and Army High Performance Computing Research Center, University of Minnesota.

[9] R. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. on Prog. Languages and Systems*, 501–538, 1985.

[10] P. Heidelberger, A. Norton, and J.T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.

[11] D. Hillis and G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29 (12):1170–1183, 1986.

[12] M. A. Huang. Solving some graph problems with optimal or near optimal speedup on mesh-of-trees networks. In *Proceedings of 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 232–240, 1985.

[13] J. Jenq and S. Sahni. All Pairs Shortest Paths on a Hypercube Multiprocessor. In *International Conference on Parallel Processing*, pages 713–716, 1987.

[14] Vipin Kumar and V. Nageshwara Rao. Load balancing on the hypercube architecture. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 603–608, 1989.

[15] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: analysis. *International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[16] Vipin Kumar and V. Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results. In *Proceedings of the International Conference on Parallel Processing*, August 1990. Extended version available as a technical report from the department of computer science, University of Minnesota, Minneapolis, MN 55455 and as MCC TR ACT-OODS-058-90.

[17] Manfred Kunde. Routing and Sorting on Mesh-Connected Arrays. In *Proceedings of the 1988 AWOC Conference*, pages 423–433, 1988. Also available as Springer LNCS Vol. 319.

[18] Hans-Werner Lang, Manfred Schimmler, Hartmut Schmeck, and Heiko Schroder. Systolic sorting on a mesh-connected network. *IEEE Transactions on Computers*, c-34(7):652–658, July 1985.

[19] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. In *Proceedings of International conference on Parallel Processing*, pages 699–706, 1987.

[20] K. Murakami, T. Kakuta, R. Onai, and N. Ito. Research on parallel machine architecture for fifth-generation computer systems. *IEEE Computer*, 18(6):76–92, 1985.

[21] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, NewYork, 1987.

[22] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, 1990.

[23] C.P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. In *Proceedings STOC*, pages 255–263, 1986.

[24] E. Shapiro, editor. *Concurrent Prolog*, chapter 7, pages 207–242. Volume 1, MIT Press, 1987.

[25] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. *Scalability of parallel sorting on mesh multicomputers*. Technical Report ACT-SPA-298-90, Microelectronics and Computer Technology Corp., Austin,TX, 1990. Also available as a technical report (number TR 90-45) from the department of computer science, University of Minnesota, Minneapolis, MN 55455.