# On the Versatility of Parallel Sorting by Regular Sampling

*Xiaobo Li*
*Paul Lu*
*Jonathan Schaeffer*
*John Shillington*
*Pok Sze Wong*

Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

*Hanmao Shi*

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

**ABSTRACT**

Parallel sorting algorithms have already been proposed for a variety of multiple instruction streams, multiple data streams (MIMD) architectures. These algorithms often exploit the strengths of the particular machine to achieve high performance. In many cases, however, the existing algorithms cannot achieve comparable performance on other architectures. Parallel Sorting by Regular Sampling (PSRS) is an algorithm that is suitable for a diverse range of MIMD architectures. It has good load balancing properties, modest communication needs and good memory locality of reference. If there are no duplicate keys, PSRS *guarantees* to balance the work among the processors within a factor of two of optimal in theory, regardless of the data value distribution, and within a few percent of optimal in practice. This paper presents new theoretical and empirical results for PSRS. The theoretical analysis of PSRS is extended to include a lower bound and a tighter upper bound on the work done by a processor. The effect of duplicate keys is addressed analytically and shown that, in practice, it is not a concern. In addition, the issues of oversampling and undersampling the data are introduced and analyzed. Empirically, PSRS has been implemented on four diverse MIMD architectures and a network of workstations. On all of the machines, for both random and application-generated data sets, the algorithm achieves good results. PSRS is not necessarily the *best* parallel sorting algorithm for any specific machine. But PSRS will achieve *good* performance on a wide spectrum of machines before any strengths of the architecture are exploited.

# 1 Introduction

Sorting is a problem of fundamental interest in computing science. For $n$ data items, sequential comparison-based sorting has a time complexity of $\Omega(n \log n)$ and mergesort, for example, is $O(n \log n)$, leaving no possibility for substantial improvements in sequential algorithms. However, significant improvements may be possible through parallelism. Many innovative MIMD parallel sorting algorithms have been proposed. In particular, extensive research has been done with sorting on hypercubes ([1, 4, 10, 14, 17, 18, 19], for example), shared memory architectures ([5, 12, 21], for example) and networks of workstations ([13, 20], for example). These algorithms often exploit the strengths of the architecture, while trying to minimize the effects of the weaknesses. Unfortunately, these algorithms usually do not generalize well to other MIMD machines.

Parallel Sorting by Regular Sampling (PSRS) is a new MIMD parallel sorting algorithm [15, 16]. The *regular sampling* load balancing heuristic distinguishes PSRS from other parallel sorting algorithms. Regular sampling has good analytical and empirical properties. Also, PSRS has modest communication needs and exhibits good per task locality of reference, reducing memory and communication contention. In theory, if there are no duplicate data values, PSRS *guarantees* to distribute the work among processors within a factor of two of optimal load balancing, regardless of the data value distribution. In practice, PSRS achieves near-perfect load balancing.

This paper makes three important contributions to our understanding of PSRS:

1. **New analytical results.** Given $p$ processors and $n$ keys to sort, the ideal case is that each processor works on $\frac{n}{p}$ data items. It has been proven that in PSRS, no processor has to work on more than $\frac{2n}{p}$ data items if $n \geq p^3$, assuming no duplicate keys [15, 16]. In this paper, a lower bound ($L = \frac{n}{p^2} + p - 1$) and a tighter upper bound ($U = \frac{2n}{p} - L$) on the amount of work done by each processor is proven. Other results on the load balancing bounds are also presented.

2. **Addresses the duplicate keys issue.** Some algorithms depend on having an uniform data value distribution to achieve good load balancing ([10], for example), or restrict the data to contain no duplicates[1] ([1, 22], for example). PSRS can handle arbitrary data value distributions and further analysis shows that the presence of duplicates increases the upper bound on load balancing *linearly*. If a single key can be duplicated a maximum of $d$ times ($d = 0$ implies no duplicates), then the upper bound on the number of data items a processor must work with becomes $U' = U + d$. If more than one data value is duplicated, then $d$ represents the key with the most number of duplicates. Since the $\frac{2n}{p}$ term dominates the bound, duplicates do not cause problems until individual keys are repeated $O(\frac{n}{p})$ times. This point is illustrated empirically by running PSRS on some application-generated data (the IMOX data set, utilized extensively in image processing and pattern recognition [2]) that contains a high percentage of duplicate items.

3. **New empirical results.** The original implementation of the algorithm was on a virtual memory, demand-paged Myrias SPS-2 with 64 processors. It achieved a 44-fold

[1] It is assumed that there is no computationally inexpensive way of removing duplicates. Techniques such as adding secondary keys may require substantial modifications to the data.

speedup while sorting 8,000,000 four-byte integers [15, 16]. This paper describes the performance of PSRS on a BBN TC2000 (shared memory), an Intel iPSC/2-386, an iPSC/860 (distributed memory with hypercube interconnections), and a network of workstations (distributed memory with a LAN interconnection). In all cases, good speedups are reported for problems of sufficient granularity, demonstrating that PSRS can be successfully used on a variety of different MIMD machines.

Although there is a plethora of parallel sorting algorithms, few have the overall versatility of PSRS:

1. **Good analytical properties.** There is a good theoretical upper bound on the worst case load balancing.
2. **Robust on different data sets.** Arbitrary data value distributions do not cause problems. Unless individual keys are repeated $O(\frac{n}{p})$ times, theoretical and empirical results show that duplicates are not a problem.
3. **Suitable for different architectures.** It is empirically shown to be suitable for a diverse range of MIMD architectures. Although PSRS may not be the best parallel sorting algorithm for any particular MIMD architecture, it is a good algorithm for a wide spectrum of architectures.

Section 2 describes the PSRS algorithm. Section 3 provides an analysis of the algorithm, in particular its load balancing properties. Section 4 presents empirical results of implementing PSRS on 4 machines (3 different architectures), for both random and application-generated data. Section 5 provides some concluding perspectives on PSRS.

# 2 The PSRS Algorithm

PSRS is a combination of a sequential sort, a load balancing phase, a data exchange and a parallel merge. Although any sequential sorting and merge algorithm can be used, PSRS is demonstrated using quicksort[2] and successive 2-way merge. Given $n$ data items[3] (indices 1, 2, 3, ..., $n$) and $p$ processors (1, 2, 3, ..., $p$), PSRS consists of four phases. Note that this description differs slightly from that in [15, 16].

Refer to Figure 1 for an example, with $n = 36$, $p = 3$ and the keys 0, 1, 2, ..., 35. For brevity, let $\rho = \lfloor \frac{p}{2} \rfloor$ and $w = \frac{n}{p^2}$.

1. **Phase One: Sort Local Data.** Each processor is assigned a contiguous block of $\frac{n}{p}$ items. The blocks assigned to different processors are disjoint. Each processor, in parallel, sorts their local block of items using sequential quicksort.

   Begin the *regular sampling* load balancing heuristic. All $p$ processors, in parallel, select the data items at local indices $1, w+1, 2w+1, ..., (p-1)w+1$ to form a representative sample of the locally sorted block[4]. The $p^2$ selected data items, $p$ from each of $p$

[2]For $n$ data items, quicksort is $O(n \log n)$ in practice, but may have a worst case of $O(n^2)$. If this is a problem, an algorithm with a worst case of $O(n \log n)$, such as mergesort, can be used.

[3]The terms data items, data values and keys are used interchangeably

[4]Note that including the first key of the list is unnecessary. It is included to simplify the analysis, allowing each of $p$ processors to take $p$ samples, instead of $p - 1$. Including this extra sample does not affect the analysis nor alter the behavior of the algorithm.

processors, are a *regular sample* of the entire data array. The local regular samples represent the keys and their value distribution at each processor.

In Figure 1, each processor is assigned $\frac{n}{p} = 12$ contiguous keys to sort. Each processor takes three samples, at the 1st, 5th and 9th indices since $w = \frac{n}{p^2} = 4$, to form the local regular sample. Note that the distance between the indices of the samples is of fixed size.

2. **Phase Two: Find Pivots then Partition.** One designated processor gathers and sorts the local regular samples. $p-1$ pivots are selected from the sorted regular sample, at indices $p + \rho, 2p + \rho, 3p + \rho, ..., (p-1) + \rho$. Each processor receives a copy of the pivots and forms $p$ partitions from their sorted local blocks. A partition is contiguous internally and is disjoint from the other partitions.

   In Figure 1, the 9 samples are collected together and sorted. From this list (0, 3, 7, 10, 13, 16, 22, 23, 27), $p - 1 = 2$ pivots are selected. The pivots are 10 and 22 (at the 4th and 7th indices), because $\rho = \lfloor \frac{p}{2} \rfloor = 1$. All processors then create three partitions.

3. **Phase 3: Exchange Partitions.** In parallel, each processor $i$ keeps the $i$th partition for itself and assigns the $j$th partition to the $j$th processor. For example, processor 1 receives partition 1 from all of the processors. Therefore, each processor keeps one partition, and reassigns $p - 1$ partitions.

   For example, in Figure 1, processor 3 sends the list (3, 4, 5, 6, 10) to processor 1, sends (14, 15, 20, 22) to processor 2, and keeps (26, 31, 32) for itself.

4. **Phase 4: Merge Partitions.** Each processor, in parallel, merges its $p$ partitions into a single list that is disjoint from the merged lists of the other processors. The concatenation of all the lists is the final sorted list.

   Note that in Figure 1, the keys in the final merged partitions at each processor are also partitioned by the pivots 10 and 22. The final sorted list is distributed over the 3 processors.

On a distributed memory MIMD architecture, information is communicated with messages: the local regular samples from Phase 1 ($p$ messages of size $O(p)$), the pivots of Phase 2 ($p$ messages of size $O(p)$) and the partitions of Phase 3 ($p$ processors sending $p-1$ messages of size $O(\frac{n}{p})$). On a shared memory architecture, all information is communicated through shared memory. In particular, Phase 3 reduces to reading and writing partitions from and to shared memory.

For pseudo-code and more details, please refer to [15, 16].

Intuitively, the notion of a regular sample to estimate the value distribution of the keys is appealing. By sampling the locally sorted blocks of all the processors, and not just a subset, the entire data array is represented. By sampling after the local blocks have been sorted, the order information of the data is captured. Since the pivots, as selected in Phase 2, divide the regular sample into almost equal partitions, the pivots should also divide the entire data array into nearly equal partitions. Also, the fixed distance intervals of the regular sampling heuristic allows a formal analysis of its effectiveness, which is presented in the next section.

Given the extensive literature on parallel sorting, it is not surprising that PSRS is similar to other algorithms. For example, PSRS is similar to the *balanced bin sort* [22] and

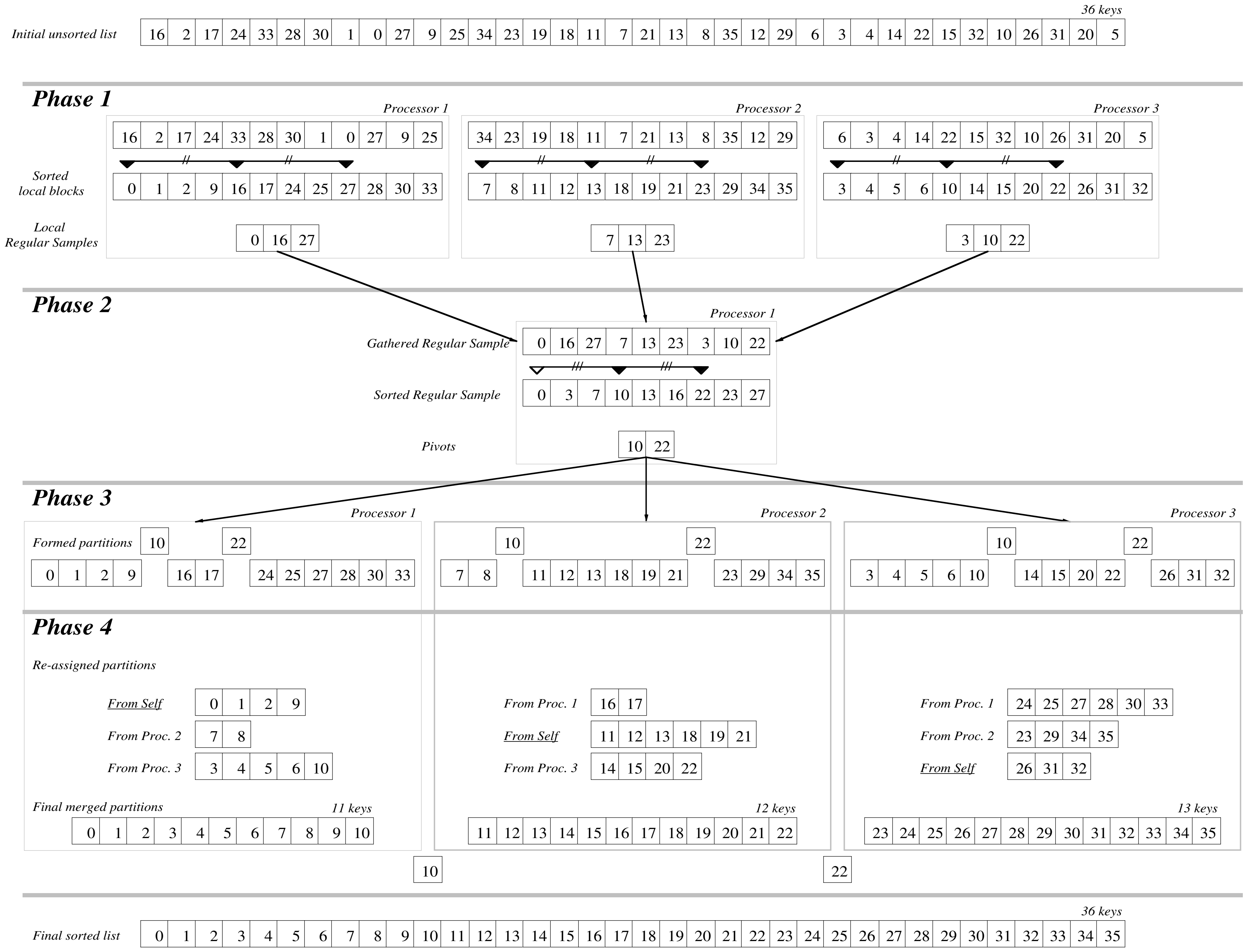



Figure 1: PSRS Example

the *load balanced sort* [1]. However, both have different approaches to load balancing than PSRS. The balanced bin sort's heuristic is also based on sampling, and results in an upper bound of $\frac{3n}{p}$ items that must be merged by a single processor. The bound is inferior to PSRS'. The load balanced sort uses an algorithm that samples and then iteratively modifies its choice of pivots until it achieves perfect load balancing. The overhead of iterating is significant because it requires additional messages and synchronization for each iteration. Also, since the algorithm iterates until perfect load balancing is achieved, duplicate keys are problematic.

The description of PSRS given above specifies that the number of samples, $s$, taken by each processor in Phase 1 is equal to $p$ (i.e. $s = p$). However, it is a natural and common extension of sampling-based algorithms to consider the techniques of *undersampling* ($s < p$) and *oversampling* ($s > p$). Intuitively, since the number of samples represents the amount of information available to the load balancing heuristic, undersampling results in poorer load balancing and oversampling results in better load balancing. These variations of the regular sampling heuristic will also be considered later on.

PSRS combines many of the successful aspects of the MIMD sorting algorithms that preceded it, and introduces the simple, but effective notion of a regular sample to help pick good pivots for the final parallel merge. The regular sample is the key, non-cosmetic difference between PSRS and similar sorting algorithms.

# 3 Time Complexity and Load Balancing

For all architectures, the time complexity of PSRS is asymptotic to $O(\frac{n}{p}\log n)$ when $n \geq p^3$, which is cost optimal [15, 16].

Another important concern in parallel sorting is load balancing. In Phase 1, the workload is evenly distributed to all processors. Phase 2 is largely sequential. Phase 3 depends on the communication properties of the computer. This section concentrates on the load balancing issue in Phase 4. A lower bound and a tighter upper bound on the number of data items each processor must merge are presented. The analysis is a function of the number of data items to be sorted, the number of processors used, the sample size and the number of times a data item is duplicated.

The keys to be sorted are the set $\{X_k\}_{k=1}^{n}$. There are at most $d+1$ duplicates of any distinct value, $d \geq 0$; i.e., for any $X_k$, there are at least $n-d-1$ other $X_l$ values with $X_l \neq X_k$. The number of processors used is $p$, $p \geq 2$ and $p$ is usually a power of 2. The quantity $\lfloor \frac{s}{2} \rfloor$ is useful and is denoted by a shorthand notation $\sigma$. In Phase 1, each processor takes $s$ local regular samples, and when gathered and sorted, the regular samples form the ordered set $\{Y_j\}$. In essence, each sample $Y_j$ represents a group of $(w-1)$ $X$ elements whose keys are greater than $Y_j$ but less than the next sample, $Y_{j+1}$, with $w = \frac{n}{ps}$. For convenience, it is assumed that $(ps)|n$ and $w \geq 2$. In Phase 4, processor $i$ will merge $\Phi_i$ $X$ elements, which are the elements between the $(i-1)$th pivot and the $i$th pivot. $N(cond)$ denotes the number of $X$ elements satisfying a certain condition *cond*. For example, $N(\leq Y_{(i-1)s+\sigma})$ represents the size of the set $\{X_k | X_k \leq Y_{(i-1)s+\sigma}\}$.

For easy reference, the above notations are summarized below.

| | |
|---|---|
| $\{X_k\}_{k=1}^{n}$ | keys to be sorted |
| $\{Y_j\}_{j=1}^{s*p}$ | regular samples, $\{Y_j\} \subseteq \{X_k\}$ |
| $s$ | the number of sample $Y$'s taken by *each* processor in Phase 1 |
| $p$ | the number of processors used |
| $d$ | the number of duplicates for the key with the most duplicates |
| $\sigma$ | $\lfloor \frac{s}{2} \rfloor$ |
| $w$ | $\frac{n}{ps}$, it is assumed that $(ps)\|n$ and $w \geq 2$ |
| $\Phi_i$ | the number of $X$ elements merged by processor $i$ in Phase 4 |
| $N(cond)$ | the number of $X$ elements satisfying a certain condition *cond* |
| $\eta$ | $\frac{1}{\eta}$ is the undersampling factor and $s = \frac{p}{\eta}$ |

The bounds for $\Phi_i$, the number of keys to merge per processor in Phase 4 is derived, in the general case. The numerical example given in Section 2 is for a special case where $d = 0$ and $s = p$.

All the $X$ elements to be merged in Phase 4 by processor $i$ must be greater than $Y_{(i-1)s+\sigma}$ and less than or equal to $Y_{is+\sigma}$.

**Lemma 1**:

Consider the $X$ elements which are less than or equal to $Y_{(i-1)s+\sigma}$, which is a pivot selected in Phase 2, for processor $i$, where $1 \leq i \leq p$:

$$N(\leq Y_{(i-1)s+\sigma}) \begin{cases} \geq w((i-1)s+\sigma-p)+p & \text{if } (i-1)s+\sigma \geq p, \\ \geq (i-1)s+\sigma & \text{otherwise} \end{cases}$$

*Proof:*

There are $(i-1)s+\sigma$ samples less than or equal to $Y_{(i-1)s+\sigma}$, and each of these samples is within a group of $w$ $X$ elements. If $(i-1)s+\sigma \geq p$, there are at most $p$ groups each with $(w-1)$ $X$ elements greater than $Y_{(i-1)s+\sigma}$. Therefore,

$$N(\leq Y_{(i-1)s+\sigma}) \geq ((i-1)s+\sigma)w - p(w-1) = w((i-1)s+\sigma-p)+p.$$

If $(i-1)s+\sigma < p$, there are at least $(i-1)s+\sigma$ samples less than or equal to $Y_{(i-1)s+\sigma}$, i.e.,

$$N(\leq Y_{(i-1)s+\sigma}) \geq (i-1)s+\sigma.$$

□

**Lemma 2**:

Consider the $X$ elements which are greater than $i$th pivot selected in Phase 2, $Y_{is+\sigma}$, for processor $i$, where $1 \leq i \leq p$:

$$N(> Y_{is+\sigma}) \geq w((p-i)s-\sigma+1)-1-d.$$

*Proof:*

Consider the case of $d = 0$. There are $sp-(is+\sigma) = (p-i)s-\sigma$ samples greater than $Y_{is+\sigma}$, i.e., $w((p-i)s-\sigma)$ elements of $X$ greater than $Y_{is+\sigma}$. There are $w-1$ elements of $X$

from the same processor as $Y_{is+\sigma}$ immediately following $Y_{is+\sigma}$ in the sorted list. These items are also greater than $Y_{is+\sigma}$. Therefore,

$$N(> Y_{is+\sigma}) \geq w((p-i)s - \sigma) + (w-1) = w((p-i)s - \sigma + 1) - 1.$$

Consider the case of $d > 0$ duplicates of $Y_{is+\sigma}$:

$$N(> Y_{is+\sigma}) \geq w((p-i)s - \sigma + 1) - 1 - d.$$

$\square$

Based on the above lemmas, the bounds for $\Phi_i$ are derived for three cases of $i$ ($i = 1$, $1 < i < p$, and $i = p$). The upper and lower bounds in these cases are denoted by $U_1$, $L_1$, $U_i$, $L_i$, $U_p$ and $L_p$, respectively. The overall bounds are denoted by $U$ and $L$.

1. Upper bound for $i = 1$:
   All the $X$ elements to be merged by processor 1 must be less than or equal to $Y_{s+\sigma}$. By Lemma 2,

   $$N(> Y_{s+\sigma}) \geq w((p-1)s - \sigma + 1) - 1 - d = n - ws - w\sigma + w - 1 - d$$

   and,

   $$\Phi_1 = n - N(> Y_{s+\sigma}) \leq U_1 = w(s + \sigma - 1) + 1 + d.$$

2. Lower bound for $i = 1$:
   By Lemma 1,
   if $s + \sigma \geq p$, then $\Phi_1 = N(\leq Y_{(2-1)s+\sigma}) \geq L_1^* = w(s + \sigma - p) + p$;

   if $s + \sigma < p$, then $\Phi_1 = N(\leq Y_{(2-1)s+\sigma}) \geq L_1^{**} = s + \sigma$.

3. Upper bound for $i = p$:
   All the $X$ elements to be merged by processor $p$ must be greater than $Y_{(p-1)s+\sigma}$. Since $(p-1)s + \sigma > p$, by Lemma 1,

   $$N(\leq Y_{(p-1)s+\sigma}) \geq w((p-1)s + \sigma - p) + p$$

   and

   $$\Phi_p = n - N(\leq Y_{(p-1)s+\sigma}) \leq U_p = w(s - \sigma + p) - p.$$

4. Lower bound for $i = p$:
   By Lemma 2,

   $$\Phi_p = N(> Y_{(p-1)s+\sigma}) \geq L_p = w(s - \sigma + 1) - 1 - d.$$

5. Upper bound for $1 < i < p$:

   $$\Phi_i = n - N(\leq Y_{(i-1)s+\sigma}) - N(> Y_{is+\sigma}).$$

   If $s \geq p$, i.e., $(i-1)s + \sigma \geq p$, by Lemma 1 and Lemma 2,

   $$\Phi_i \leq U_i = U_i^* = n - (ps - s - p + 1)w - p + 1 + d = w(s + p - 1) - p + 1 + d.$$

If $s < p$ but $(i-1)s + \sigma \geq p$, $\Phi_i \leq U_i^*$.

If $s < p$ and $(i-1)s + \sigma < p$,

$$\Phi_i \leq U_i = U_i^{**} = n - ((i-1)s + \sigma) - (w((p-i)s - \sigma + 1) - 1 - d),$$

which depends on $i$.

In the case of $s = p/\eta$ for integer $\eta \geq 2$, for $2 \leq i \leq \eta$, we have $(i-1)s + \sigma < p$, thus $U_\eta = w(\eta s + \sigma - 1) - (\eta - 1)s - \sigma + 1 + d$. For $i \geq \eta + 1$, we have $(i-1)s + \sigma \geq p$, thus $\Phi_i \leq w(s + p - 1) - p + 1 + d$.

6. Lower bound for $1 < i < p$:
There are $s$ samples which are greater than $Y_{(i-1)s+\sigma}$ and less than or equal to $Y_{is+\sigma}$. If $s \geq p$, there are $(s-p)(w-1)$ elements of $X$ must be merged by processor $i$. There are $w-1$ elements of $X$ from the same processor as $Y_{(i-1)s+\sigma}$ immediately following $Y_{(i-1)s+\sigma}$ in the sorted list. These items are also greater than $Y_{(i-1)s+\sigma}$. Therefore,

$$\Phi_i \geq L_i = L_i^* = s + (s-p)(w-1) + (w-1) = w(s-p+1) + p - 1.$$

If $s < p$, $\Phi_i \geq L_i = L_i^{**} = s$.

The overall bounds $U$ and $L$ are derived in three cases of sampling: $s = p$, $s \geq p$ and $s < p$. When $s \geq p$, since $w\sigma > w - 1$, then $U_p < U_i$ and $L_1 > L_i$, so $U$ could only be either $U_1$ or $U_i$, and $L$ could only be either $L_p$ or $L_i$. Using the above results, the following theorems present the bounds in overall case, $U$ and $L$.

**Theorem 1** (Load Balancing Bounds for No Duplicates):

When $s = p$ and $d = 0$:

$$L = \tfrac{n}{p^2} + p - 1$$

$$U = 2(\tfrac{n}{p}) - \tfrac{n}{p^2} - p + 1 = 2(\tfrac{n}{p}) - L$$

*Proof:*

Since $w\sigma < wp - p$, then $U_1 < U_i$ and $L_i < L_p$, thus $L = L_i = w + p - 1 = \frac{n}{p^2} + p - 1$ and $U = U_i = 2wp - w - p + 1 = 2(\frac{n}{p}) - L = 2(\frac{n}{p}) - \frac{n}{p^2} - p + 1$. $\square$

Note that when $s = p$ and $d \geq 0$,

$$U = 2(\frac{n}{p}) - \frac{n}{p^2} - p + 1 + d.$$

**Theorem 2** (Load Balancing Bounds for Duplicates and Oversampling):

In the case with oversampling ($\sigma = \lfloor \frac{s}{2} \rfloor \geq p$) and duplicates ($d \geq 0$):

For $1 < i < p$,

$$U_i = \tfrac{n}{p} + \tfrac{n}{s}(1 - \tfrac{1}{p}) - p + 1 + d,$$

$$L_i = \tfrac{n}{p} - \tfrac{n}{s}(1 - \tfrac{1}{p}) + p - 1.$$

For $1 \le i \le p$,

$$U = (\frac{3}{2})\frac{n}{p} - \frac{n}{ps} + 1 + d,$$

$$L = (\frac{1}{2})\frac{n}{p} + \frac{2n}{ps} - 1 - d.$$

*Proof:*

Since $\sigma \ge p$, then $U_1 > U_i$ and $L_p < L_i$, but

$$U_1 = \frac{n}{p} + \frac{n}{ps}\lfloor\frac{s}{2}\rfloor - \frac{n}{ps} + 1 + d \le \frac{n}{p} + \frac{n}{ps}(\frac{s}{2}) - \frac{n}{ps} + 1 + d = (\frac{3}{2})\frac{n}{p} - \frac{n}{ps} + 1 + d$$

and

$$L_p = \frac{n}{p} - \frac{n}{ps}\lfloor\frac{s}{2}\rfloor + \frac{n}{ps} - 1 - d \ge \frac{n}{p} - \frac{n}{ps}(\frac{s}{2} - 1) + \frac{n}{ps} - 1 - d = (\frac{1}{2})\frac{n}{p} + \frac{2n}{ps} - 1 - d.$$

$\square$

Several conclusions can be drawn from the above theorems for the case $s \ge p$:

1. Each processor will merge at least $w = \frac{n}{ps}$ and at most $2(\frac{n}{p}) - w$ elements. If oversampling is used ($s > p$), the upper bound $U$ becomes $(\frac{3}{2})\frac{n}{p} - \frac{n}{ps} + 1 + d$, which is dominated by the $(\frac{3}{2})\frac{n}{p}$ term, since the value of $d$ is much smaller than $\frac{n}{p}$ in practice.
2. Duplicates increase upper bounds and reduce lower bounds **linearly**, and will cause only a minor performance degradation, unless an item is duplicated $O(\frac{n}{p})$ times.
3. Oversampling improves the bounds for $1 < i < p$, but not for $i = 1$ and $i = p$. Thus oversampling should improve the load balancing in most cases. The gains in Phase 4 from oversampling may be offset by the increased cost of Phase 2.

The case of undersampling ($s < p$) is more complex. Only the upper bound for $\Phi_i$ is given to show the effect of undersampling on load balancing, for the case $s = \frac{p}{\eta}$ for integer $\eta \ge 2$ and $\frac{1}{\eta}$ is the undersampling factor.

Either $U_\eta$ or $U_p$ could be larger depending on the value of $d$, which is stated in the following theorem.

**Theorem 3** (Load Balancing Bounds for Duplicates and Undersampling):

If $d \ge 0$ and $s = \frac{p}{\eta}$, then $U$ could be either

$$U_\eta = (\eta + \frac{1}{2})(\frac{n}{p}) - \frac{\eta n}{p^2} - \frac{2\eta - 1}{2\eta}p + 1 + d, \text{ or}$$

$$U_p = (\eta + \frac{1}{2})(\frac{n}{p}) - p.$$

For example, when $s = \frac{p}{2}$, $U$ could be either

$$U_2 = (\frac{5}{2})(\frac{n}{p}) - \frac{2n}{p^2} - \frac{3}{4}p + 1 + d, \text{ or}$$

$$U_p = (\frac{5}{2})(\frac{n}{p}) - p.$$

When $s = \frac{p}{4}$, $U$ could be either

$$U_4 = (\frac{9}{2})(\frac{n}{p}) - \frac{4n}{p^2} - \frac{7}{8}p + 1 + d, \text{ or}$$

$$U_p = (\frac{9}{2})(\frac{n}{p}) - p.$$

It can be concluded for the undersampling case $s = \frac{p}{\eta}$ that:

1. In the case of undersampling, the workload balancing in Phase 4 gets worse.

2. The upper bound of the workload in Phase 4 increases linearly with $\eta$. It is approximately $\eta + \frac{1}{2}$ times the optimal $(\frac{n}{p})$.

3. Undersampling is justified only if the time saving in Phase 2 is significant.

# 4 Empirical Analysis

The theoretical results for PSRS discussed in the last section are encouraging. The upper bounds on the load balancing are good, but how realistic are they in practise? In this section, PSRS is analyzed from an empirical point of view. Implementations of PSRS on several different MIMD machines are examined. Both uniformly distributed and nonuniformly distributed data sets are considered.

## 4.1 Implementations

In addition to the original Myrias SPS-2 implementation, PSRS has been implemented on a diverse range of MIMD computers: a BBN TC2000 (interleaved shared memory), Intel iPSC/2-386 and iPSC/860 hypercubes (distributed memory with hypercube interconnections) and a local area network (LAN) of workstations (distributed memory with a LAN interconnection). The number of processors available and the amount of memory per processor varies for the different machines. Consequently, the number of experimental data points for each machine also varies. All four implementations are similar to the Myrias version of PSRS, with the exception that the methods of communication and synchronization are machine dependent. All of the programs are written in C.

The relative cost of communication most distinguishes the different MIMD machines. It varies from expensive (message passing on a single shared bus) to relatively inexpensive (shared memory). It should be noted that the TC2000, iPSC/2-386 and iPSC/860 all have communication hardware whose bandwidth increases as the number of processors increases. Regardless of the number of processors used for our LAN of workstations, there is always just one physical wire and the LAN's bandwidth remains constant. For this reason, the LAN results reflect greater communication overheads as the problem size and and the number of processors increase.

Little effort has been made to tailor the program to any of the targeted architectures. There are several machine and implementation-specific factors that could quickly turn our experiment into a programming exercise. The goal is to investigate the potential performance of the algorithm in general, and not to benchmark specific machines or implementations. Therefore, none of the implementations are, in our opinion, optimized to the hardware.

The BBN TC2000 is a tightly coupled MIMD multiprocessor with both shared and local memory. Each processor is a Motorola 88100 RISC processor. Although there are 16 megabytes of RAM physically local to each processor, it is segmented into both shared and private portions. The shared memory portions on all processor boards are mapped into one continuous address space by the memory management hardware. Communication between processor boards is via a high bandwidth, multistage network based on the Butterfly switch. A reference to a memory location that is not physically local to the processor causes a memory access request to be sent to the processor board that does contain the data. In this way, memory is transparently shared among the processors at the application code level.

In the TC2000 implementation, the data array is kept in shared memory. However, local memory is used to cache each processor's disjoint piece(s) of the data array in Phases 1 and 4. Since accessing shared memory is more expensive than local memory, the memory access intensive operations of sequential sorting and merging are performed in local memory. In effect, shared memory is only used to communicate samples, pivots and data partitions. The program uses the PCP pre-processor to express the parallelism [6, 8].

The iPSC (Intel Personal Super Computer) is a family of loosely-coupled distributed memory multiprocessors based on the hypercube architecture. Each node of the iPSC/2-386 contains an Intel 80386 processor and 8 megabytes of RAM. Communication between nodes is done via message passing over the hypercube nearest neighbor links. The iPSC/860 uses an Intel i860 processor at each of the hypercube nodes instead of the 80386. Since the memory of the iPSC is distributed, all sorting and merging is performed in local memory. Data samples, pivots and data partitions are communicated between processors by messages. The program uses the standard Intel message passing libraries [9].

The LAN implementations of PSRS use Sun 4/20 workstations connected by a single Ethernet with 10 megabit/second bandwidth. There are two different LAN implementations, each using a different message passing communications package. The first implementation uses the Network Multiprocessor Package (NMP) [11], a locally produced library that provides a friendly high-level interface to sockets and TCP/IP. For large sorting problems, it was discovered that the partitions in Phase 3 fill all of the available message buffers, resulting in deadlock. Each processor, by default, has only 4K bytes of message buffers, which is insufficient for the volume of communication in Phase 3. The buffer size can be increased, but only to a system limit of 52K bytes. For sufficiently large problems, the program still deadlocks. To avoid this, the data is packaged into smaller pieces and is then sent using multiple synchronized messages. Obviously, the extra synchronization adversely affects performance.

The second implementation uses the ISIS package from Cornell (version 2.1) [3]. It is claimed that ISIS provides communication performance as fast as remote procedure calls. However, our version of ISIS requires 30% of each packet as overhead for control information, thus increasing the number of packets needed and decreasing communication throughput performance. Programs written in ISIS execute in the form of tasks/threads. Also, the communication protocols of ISIS are timer based. Extra time is required in the communication protocol, memory management and tasking layers of ISIS. These considerations restrict the speedups obtainable for PSRS.

Of the two implementations, the NMP-based version provide the better results and they are reported here. Although the speedups are respectable for a LAN-based MIMD machine, they should be viewed as lower bounds since an awkward work-around for the limited buffer sizes of our system is required.

## 4.2 Performance Measures and Sequential Timings

Speedup is a measure of how much faster a task can be performed with many processors compared with one processor:

$$S_p = T_1/T_p$$

where $S_p$ is the speedup for $p$ processors, $T_1$ is the sequential processing time, and $T_p$ is the processing time with $p$ processors.

Speedup efficiency is a measure of how the observed speedup compares with linear speedup, which is ideal. It is defined as:

$$E_p = S_p/p$$

Implicitly, efficiency measures the portion of time each processor spends doing "useful" work that contributes to the final solution.

Because of the large size of some of the data sets, it is impossible to record some of the single processor times. In these situations, the single processor time is extrapolated based on the known time of the largest array that could be sorted in local memory without paging. The following formula is used to extrapolate the times for each machine:

$$T_1(n) = \frac{n \log n}{1,000,000 \log 1,000,000} \times T_1(1,000,000)$$

where $T_1(n)$ is the extrapolated time for 1 processor sorting $n$ elements, and $T_1(1,000,000)$ is the measured time required for 1 processor to sort 1,000,000 elements.

## 4.3 Experiments Using Random Data

Experiments sorting arrays of integers created with a pseudo-random number generator are performed. The values are uniformly distributed in the range 0 through $2^{32} - 1$. Each data point reported in the tables below represent the average results over five different sets of random data. Each set of data is created using a different random number generator seed value. No tests are made for duplicate data items, of which there were undoubtably a few. Unless otherwise stated, the sample size is $p$.

As is the convention in the literature, the data to be sorted is already distributed among the processors before the timing is begun. After the timing begins, the processors proceed with the sequential quicksort of Phase 1. When all of the processors have finished their merge of Phase 4, the sort is considered complete and the timing ends. For distributed memory machines, the final sorted data array remains distributed among the different processors at the end of the sort. The concatenation of the memories of the different processors is considered to be the final sorted array. For the shared memory BBN TC2000, the final sorted data array is in shared memory.

During the timings, no other processes or users contended for the processors, except for the LAN results, which were run with a minimal amount of contention from other processes. Figures 2, 3, 4 and 5 show the speedups PSRS achieves on the BBN TC2000, iPSC/2-386, iPSC/860 and a LAN of workstations respectively. The keys are randomly generated and uniformly distributed. Tables 1, 2, 3 and 4 show the corresponding real execution times. Note that the number of processors available varies between the machines.

The BBN TC2000 speedups in Figure 2 reinforce the positive conclusions from previous experiments with the Myrias SPS-2 [16].

| Sizes | Sorting Times (in seconds) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1PE | 2PEs | 4PEs | 8PEs | 16PEs | 32PEs | 64PEs |
| 100,000 | 1.29 | 1.04 | 0.54 | 0.28 | 0.17 | 0.16 | - |
| 200,000 | 2.71 | 2.22 | 1.12 | 0.60 | 0.32 | 0.24 | - |
| 400,000 | 5.81 | 4.56 | 2.36 | 1.22 | 0.65 | 0.41 | - |
| 800,000 | 15.97 | 9.49 | 4.85 | 2.51 | 1.31 | 0.75 | 0.75 |
| 1,000,000 | 22.15 | - | 6.15 | 3.16 | 1.66 | 0.93 | 0.85 |
| 2,000,000 | 46.52 | - | - | - | 3.41 | 1.83 | 1.32 |
| 4,000,000 | 97.49 | - | - | - | - | 3.68 | 2.31 |
| 8,000,000 | 203.86 | - | - | - | - | 7.47 | 4.29 |

Table 1: Sorting times for BBN TC2000, uniform distribution

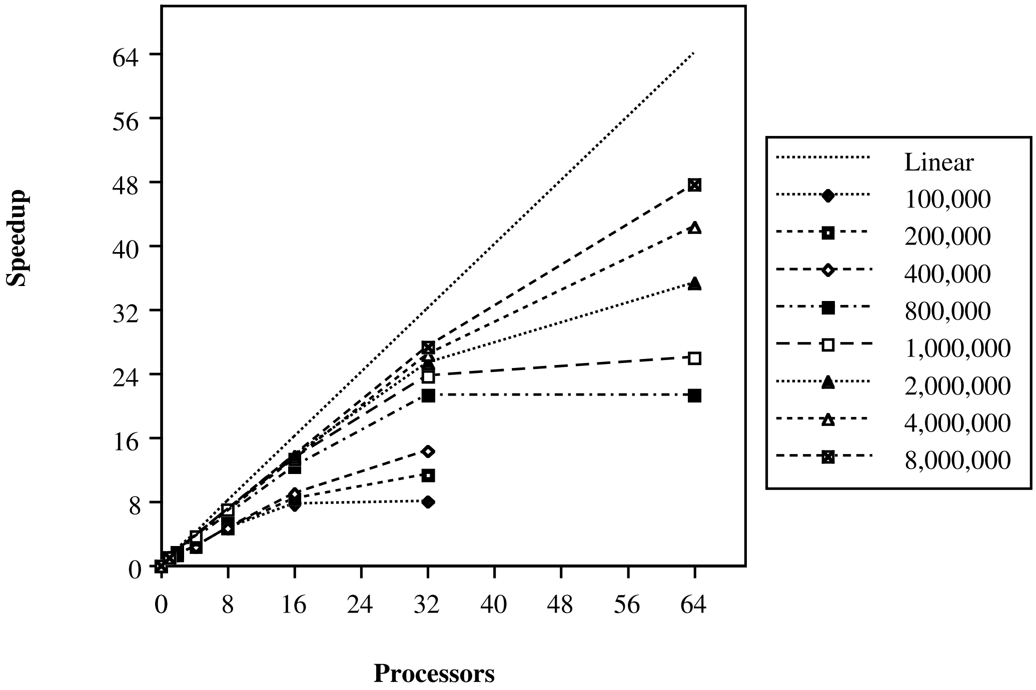



Figure 2: Speedups for BBN TC2000, uniform distribution

| Sizes | Sorting Times (in seconds) | | | | | |
|---|---|---|---|---|---|---|
| | 1PE | 2PEs | 4PEs | 8PEs | 16PEs | 32PEs |
| 100,000 | 8.43 | 4.67 | 2.28 | 1.26 | 0.66 | 0.49 |
| 200,000 | 17.82 | 9.81 | 4.81 | 2.61 | 1.31 | 0.84 |
| 400,000 | 37.86 | 20.60 | 10.08 | 5.44 | 2.69 | 1.56 |
| 800,000 | 79.63 | 43.20 | 21.13 | 11.36 | 5.56 | 3.08 |
| 1,000,000 | 101.17 | - | 26.82 | 14.37 | 7.04 | 3.88 |
| 2,000,000 | 212.50 | - | - | - | - | 7.88 |
| 4,000,000 | 445.30 | - | - | - | - | 16.20 |

Table 2: Sorting times for iPSC/2-386, uniform distribution

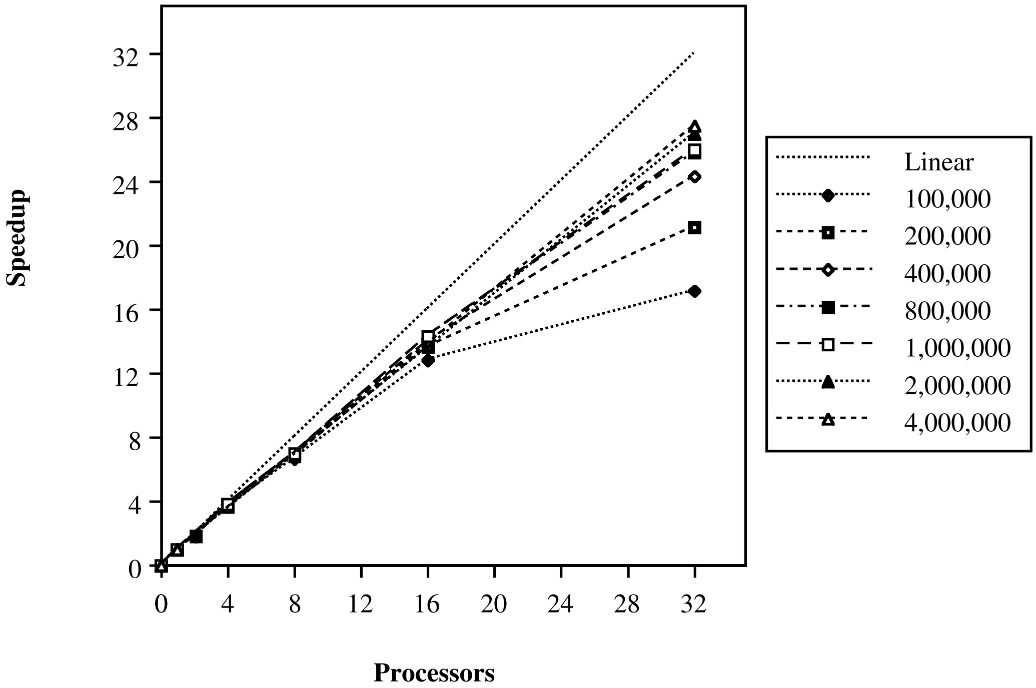



Figure 3: Speedups for iPSC/2-386, uniform distribution

| Sizes | Sorting Times (in seconds) | | | |
|---|---|---|---|---|
| | 1PE | 4PEs | 16PEs | 64PEs |
| 100,000 | 0.75 | 0.26 | 0.10 | 0.11 |
| 200,000 | 1.59 | 0.55 | 0.18 | 0.15 |
| 400,000 | 3.33 | 1.13 | 0.37 | 0.20 |
| 800,000 | 7.05 | 2.45 | 0.71 | 0.31 |
| 1,000,000 | 8.95 | 3.02 | 0.90 | 0.37 |
| 2,000,000 | 18.80 | - | 1.71 | 0.63 |
| 4,000,000 | 39.40 | - | - | 1.19 |
| 8,000,000 | 82.40 | - | - | 2.46 |

Table 3: Sorting times for iPSC/860, uniform distribution

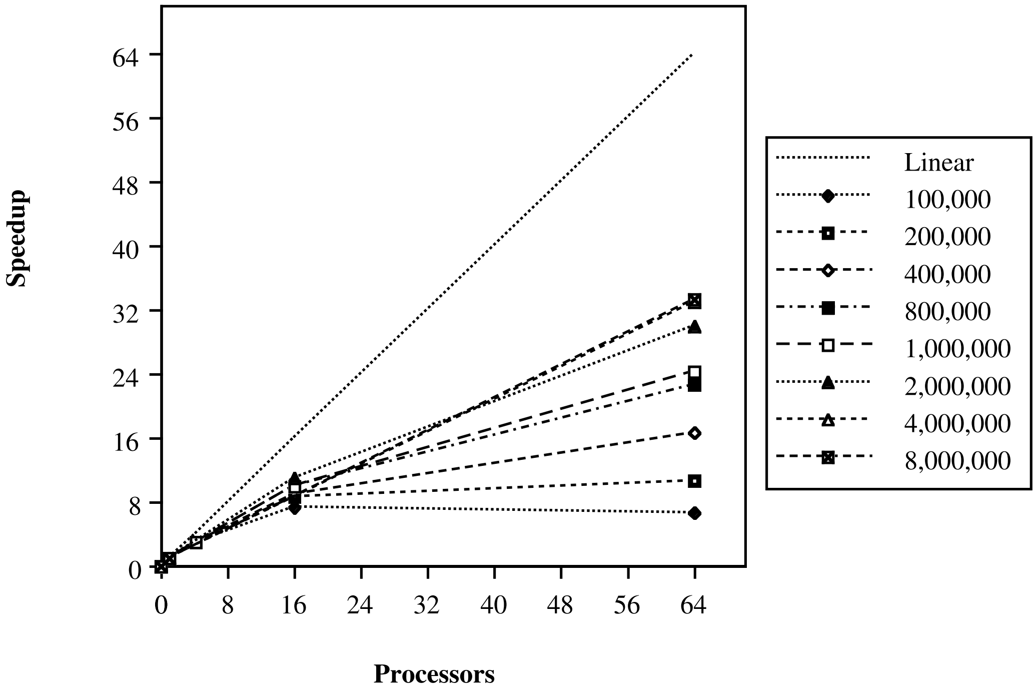



Figure 4: Speedups for iPSC/860, uniform distribution

| Sizes | Sorting Times (in seconds) | | | | |
|---|---|---|---|---|---|
| | 1PE | 2PEs | 4PEs | 8PEs | 16PEs |
| 100,000 | 4.73 | 3.11 | 1.36 | 1.13 | 0.81 |
| 200,000 | 10.02 | 6.59 | 4.10 | 1.91 | 1.57 |
| 400,000 | 21.18 | 10.85 | 8.11 | 4.87 | 3.28 |
| 800,000 | 44.63 | 26.88 | 14.74 | 10.47 | 6.33 |
| 1,000,000 | 56.70 | 36.52 | 20.60 | 15.19 | 8.69 |
| 2,000,000 | 119.09 | - | 39.88 | 32.68 | 22.13 |
| 4,000,000 | 249.56 | - | - | 84.36 | 50.93 |

Table 4. Sorting times for LAN, uniform distribution

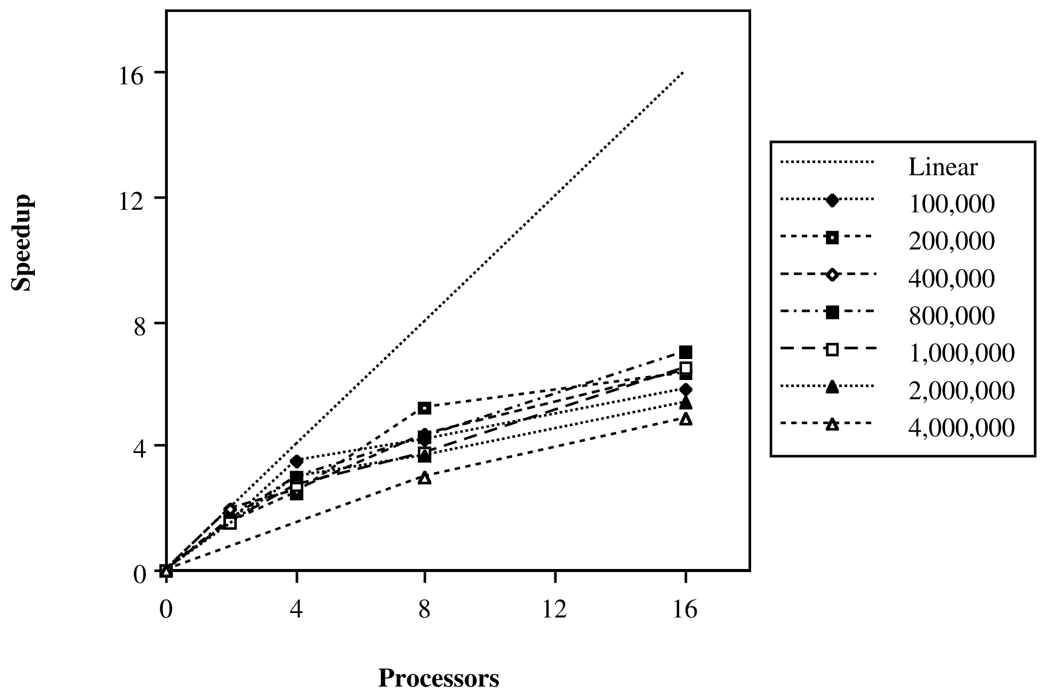



Figure 5: Speedups for LAN, uniform distribution

The notion of granularity, the amount of computation between communication and synchronization points, is important to all forms of MIMD programming. Since sequential processes do not need to communicate or synchronize, granularity is one measure of the ratio between useful computation and the overheads of parallelism. The effect of granularity on PSRS for the TC2000 can be seen in the relationship between the number of processors and the size of the sorting problem.

For a fixed number of processors, increasing the problem size increases the speedup efficiency. When using 64 processors, the speedup curve grows closer to linear as the problem size is incrementally increased from 800,000 integers to 8,000,000 integers. For 8,000,000 integers, a maximum speedup of 47.5 is observed for 64 processors. For a fixed problem size, adding more processors to the solution results in diminishing speedup returns. For 800,000 integers, the speedup remains at 16 whether using 32 processors or 64 processors. Obviously, larger data sets more effectively offset the overheads of the algorithm and of the machine through higher granularity. Most importantly, there appears to be no inherent limit on the speedup for the PSRS algorithm. As long as the problem size is of sufficient granularity, PSRS can efficiently use an arbitrary number of processors.

The notion of "sufficient granularity" is difficult to quantify and to gain agreement on. It is clear, however, that it is only of theoretical interest because the size of many problems cannot or should not be increased simply to take better advantage of additional processors. In practice, there is usually a specific problem size that must be solved. If PSRS, or another algorithm, cannot perform well on that particular problem size, then parallelism is of little use. The issue of improving speedup performance for a fixed problem size will be addressed later on.

It should be noted that a flat speedup curve for a problem size does not necessarily indicate poor performance. In particular, the speedup curve between 32 and 64 processors while sorting 800,000 integers on the TC2000 is flat, but the real time is already a low 0.75 seconds. This compares with a sequential sorting time of 15.97 seconds for the same problem size. It is unrealistic to expect a parallel algorithm to reduce real running times to arbitrarily close to zero as additional processors are added.

Overall, the iPSC/2-386 speedups are better than the TC2000, achieving a peak speedup of 27.49 on 32 processors while sorting 4,000,000 items. Using speedup efficiency as a measure of performance, this represents our best data point, with each processor spending $\frac{27.49}{32} = 86\%$ of its time doing useful work. PSRS was not tested on a larger dimension iPSC/2-386 because one was not available at the time.

The Intel hypercube has dedicated nearest neighbor communication connections. The special communication links of the hypercube allow for fast data transfer between many different processors and with different communication patterns. It is interesting to note that the more sophisticated pattern of communication in Phase 3, as described in [15, 16], was not implemented. Instead of explicitly scheduling and synchronizing the messages so as to take advantage of the dedicated hypercube links, the implementation simply iterated through all the node numbers in Phase 3. Apparently, the performance loss from ignoring the hypercube links is not large enough to spoil the speedups of our implementations.

For problems of large granularity, the speedup efficiency of the TC2000 and the iPSC/2-386 are similar. It is on the lower granularity experiments that the hypercube is clearly more efficient than the TC2000. The smaller the dimension of the hypercube, the larger the fraction of processors that can be reached by nearest neighbour links[5]. Since a nearest neighbour link is an one-stage, direct channel between hypercube nodes, it results in the fastest possible communication. Consequently, the average cost of communicating one byte is lower when the dimension of the hypercube is lower. On the TC2000, the number of stages in the network from processor to nonlocal memory is constant for the size of the full machine and does not vary as the number of processors in the experiment varies. Thus, while the communication overhead of the hypercube is lower for small numbers of processors, the communication overhead for the TC2000 remains constant. This may explain the better speedup efficiency of the hypercube for small numbers of processors.

The iPSC/860 speedups are lower than the iPSC/2-386's, achieving a maximum of 33-fold for 64 processors while sorting 8,000,000 items. However, a single i860 processor is roughly an order of magnitude faster than a single 80386, using the real time to sort 1,000,000 random integers as a guideline. A faster i860 processor implies that for a fixed hypercube and problem size, the granularity of the problem is less on the iPSC/860 than on the iPSC/2-386. Therefore, the 386-based hypercube achieves better speedup efficiency, but the i860-based hypercube achieves faster real times.

Compared to the other results, the performance of sorting on a network of workstations is markedly lower. Figure 5 shows a peak speedup of 7.1, while sorting 800,000 data items on 16 machines. Although this sounds like a poor result, it is important to keep in mind the implementation caveats mentioned previously, and the fact that communication over a LAN is expensive. The figure also illustrates that bigger data sets do not necessarily achieve better performance. In this case, the apparent anomaly is explained by the synchronization added to Phase 3. Bigger data sets require that they be broken down into more 52K byte pieces, introducing more performance loss due to synchronization overheads.

## 4.4 Load Balancing and RDFAs

One of the strengths of the PSRS algorithm is the claim of good load balancing. RDFA is a measure of how evenly a load is balanced among processors during the merge in Phase 4 [15, 16]. It stands for *R*elative *D*eviation of the size of the largest partition *F*rom the *A*verage size of the $p$ partitions, and is defined as:

$$RDFA = \frac{m}{n/p} = \frac{m \times p}{n}$$

where $m$ is the maximum number of keys merged by any processor in Phase 4, $n$ is the total number of elements to sort, and $p$ is the number of processors. Perfect load balancing will result in a RDFA of 1.0, and the RDFA will always be greater than or equal to 1.0 since $m \geq \frac{n}{p}$. If there are no duplicates in the data, Theorem 1 guarantees that the RDFA will be less than 2.0. If there are duplicates, the upper bound on the RDFA increases linearly with respect to the number of duplicates for the key with the most duplicates. Duplicates increase the upper bound on RDFA linearly and will cause only a minor performance degradation, unless a key is duplicated $O(\frac{n}{p})$ times.

Table 5 shows the set of RDFAs for the data used in the experiments. All the RDFAs are remarkably close to the optimal value of 1. In fact, the worst RDFA (1.075) is within

[5] For 4 processors, the hypercube is of dimension 2 and 1/2 of the processors are nearest neighbours to any other. For 32 processors, the hypercube is of dimension 5 and only 5/32 of the processors are nearest neighbours to any other.

| Sizes | RDFAs | | | | | |
|---|---|---|---|---|---|---|
| | 2PEs | 4PEs | 8PEs | 16PEs | 32PEs | 64PEs |
| 100,000 | 1.002 | 1.004 | 1.015 | 1.038 | 1.075 | - |
| 200,000 | 1.001 | 1.004 | 1.008 | 1.021 | 1.069 | - |
| 400,000 | 1.001 | 1.003 | 1.006 | 1.019 | 1.045 | - |
| 800,000 | 1.001 | 1.001 | 1.006 | 1.011 | 1.024 | 1.061 |
| 1,000,000 | - | 1.002 | 1.005 | 1.012 | 1.018 | 1.044 |
| 2,000,000 | - | - | - | 1.006 | 1.016 | 1.030 |
| 4,000,000 | - | - | - | - | 1.011 | 1.025 |
| 8,000,000 | - | - | - | - | 1.008 | 1.016 |

Table 5. RDFAs for uniform distribution

7.5% of optimum. Clearly, regular sampling does an excellent job of dividing the work of the parallel merge equally, given an uniformly distributed data set.

## 4.5 Algorithmic Bottleneck Analysis

The speedups achieved on the TC2000, iPSC/2-386 and iPSC/860 all improve if the problem size increases and the number of processors is held constant. Likewise, when both the problem size and the number of processors are increased, the speedups also increase. This is expected, since an increase in the granularity of work given to each processor better offsets the system's and algorithm's overheads. However, in this section, the algorithm is analyzed to identify bottlenecks that limit the ability to extrapolate the results to larger numbers of processors. In particular, when the problem size is constant but the number of processors increases, the speedup curve eventually flattens or a slowdown is observed. It has already been argued that it is unrealistic to expect any algorithm to arbitrarily approach zero real time as processors are added. However, an attempt is made to ameliorate the impact of reduced granularity.

Figure 6 shows a timing analysis for a sort of 800,000 integers on the TC2000. Figure 6a reproduces the speedup curve for the data size on the TC2000. Figure 6b shows how much absolute time was spent in Phases 1, 2 and 4 (there is no Phase 3 for the TC2000). As the number of processors increases, the number of elements each processor has to sort in Phase 1 and to merge in Phase 4 deceases ($\frac{800{,}000}{p}$). Hence, as shown in Figure 6b, the time spent in Phases 1 and 4 decreases as more processors are added.

Unfortunately, as the number of processors increases, the amount of work to be done in Phase 2 also increases. Recall that Phase 2 mainly consists of communicating samples, sorting samples and communicating pivots. The number of samples is a function of the number of processors. The absolute time spent in Phase 2 increases marginally, as seen in Figure 6b. But Figure 6c clearly shows that Phase 2 begins to dominate the running time of the algorithm in terms of its percentage of the real time. The cost of Phase 2 is shown to quickly grow from being less than 2% of the total cost with 16 processors, to 20% with 32 processors and 40% with 64 processors. Not only does the decreasing $\frac{n}{p}$ ratio contribute to lower problem granularity, but the increasing communication needs of Phase 2 adds to the single processor bottleneck.

Of course, the real times required for Phase 2 do not change when sorting larger problems with a fixed number of processors, since the amount of work in Phase 2 only depends on $p$. However, because larger problems have larger Phases 1 and 4, the percentage of real time spent in Phase 2 is less, and the impact of the bottleneck is less pronounced. In fact, reexamining the speedup curve for this data size, it can be seen that the knee in Figure 6a, where it tapers off into a flat line, coincides with the rapid growth of Phase 2's share of the total time. Amdahl's Law predicts the diminishing returns through the addition of processors to the same problem size.

Figures 7 and 8 show the same timing analysis for the iPSC/2-386 and iPSC/860, respectively. With the iPSC/2-386, the individual processors are slower than those for the TC2000. Consequently, more real time is spent in Phases 1 and 4 for the same values of $n$ and $p$, reducing the impact of Phase 2. Similarly, the cost of communicating partitions of data in Phase 3 is still not high enough to dominate the cost of Phases 1 and 4. With the iPSC/860, the faster processors reduce the granularity of Phases 1 and 4. In fact, the single largest phase for the iPSC/860, by percentage of real time, is Phase 3. It suggests that for the iPSC/860, the amount of computational power is not as evenly matched with communication power as on the TC2000 and the iPSC/2-386. Still, the Phase 2 percentage grows quickly when the number of processors is doubled from 32 to 64.

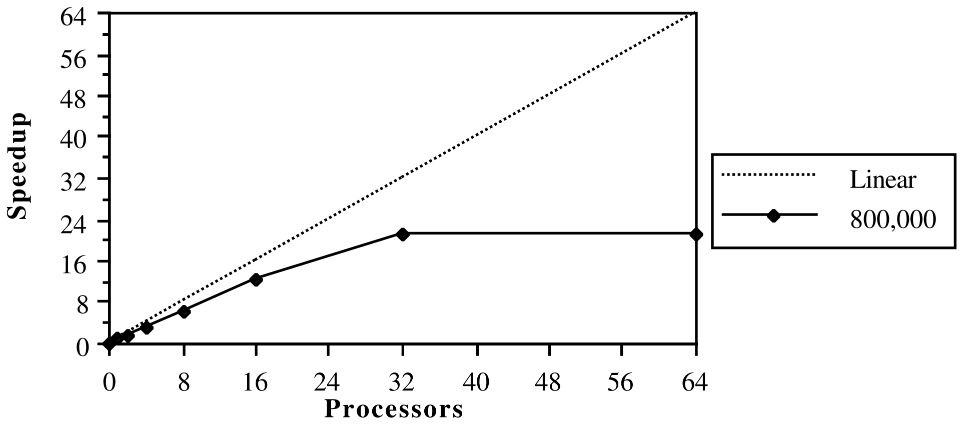
For completeness, the same timing analysis for the LAN of workstations implementation is included in Figure 9. It can be seen in Figure 9c that the percentage of real time spent in Phase 2 grows as more processors are added to the problem. But, due to the single bus and fixed bandwidth of the network, the cost of communicating partitions in Phase 3 is by far the largest fraction of PSRS. In particular, the extra synchronization required in Phase 3 skews the results, causing this phase to dominate the execution time. Again, the computational power of the processing nodes (i.e. workstations) is unevenly matched with the cost of communication. For situations where a network of workstations is the only available parallel computer, PSRS can still be effectively used when sorting large problems.

The phase-by-phase analysis reveals the important balance between processing power and communication costs. Because each of the MIMD machines has a different communication mechanism and strengths, the impact of the communication intensive Phase 3 is also different. Phase 3 is a machine dependent aspect of our experiment. However, Phase 2 is an algorithmic bottleneck, independent of any particular MIMD machine. Faster processors or cheaper communications alone cannot completely solve the Phase 2 bottleneck.

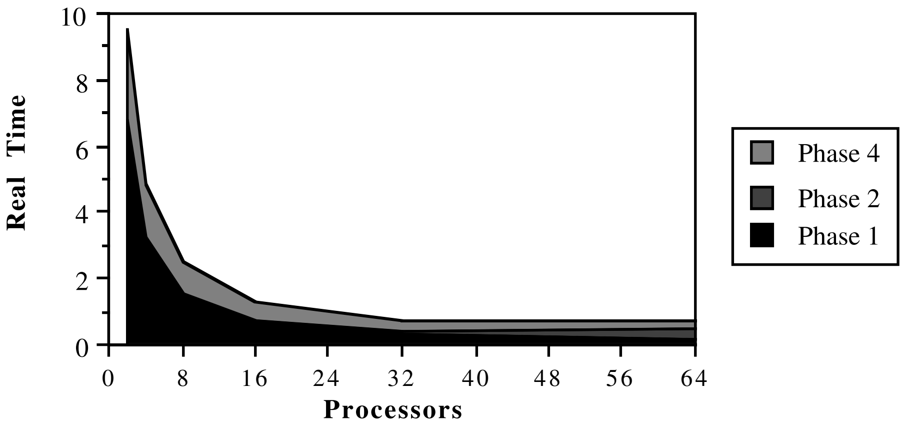
Two broad, and not fully satisfactory, approaches to dealing with the Phase 2 bottleneck are offered. First, the sorting of the samples by the lone processor in Phase 2 can itself be parallelized. Because PSRS is best suited to sorting large numbers of keys, it is probably not well suited for a parallel Phase 2. A different parallel sorting algorithm, hyperquicksort [19] for example, may be better suited for sorting the small data sets of Phase 2. Similarly, the process of gathering the samples can be implemented as a parallel binary tree merge, but at the cost of additional message and synchronization overheads.

Second, undersampling (when $s < p$) can be used to reduce the number of elements communicated and sorted in Phase 2. Undersampling lessens the impact of the Phase 2 bottleneck, but does not solve the fundamental problem. Furthermore, the theoretical results of Section 3 predict that the upper bound on worst-case load balancing in Phase 4 grows rapidly as $s$ becomes less than $p$. There is a clear tradeoff between a shorter Phase 2 and a potentially longer Phase 4 due to poorer load balancing.

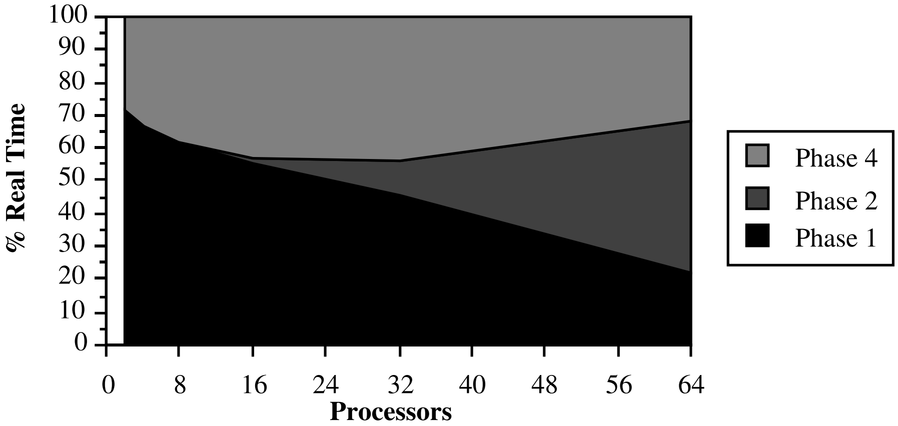
In one experiment, $s$ was set to be $0.5 * p$, an undersampling factor of 0.5, and PSRS was




(a) Speedup Curve

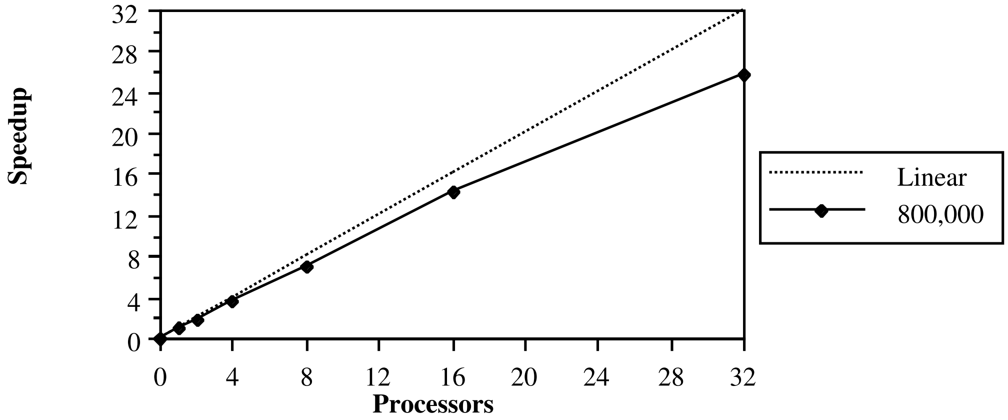



(b) Phase-by-Phase Analysis, real time (seconds)

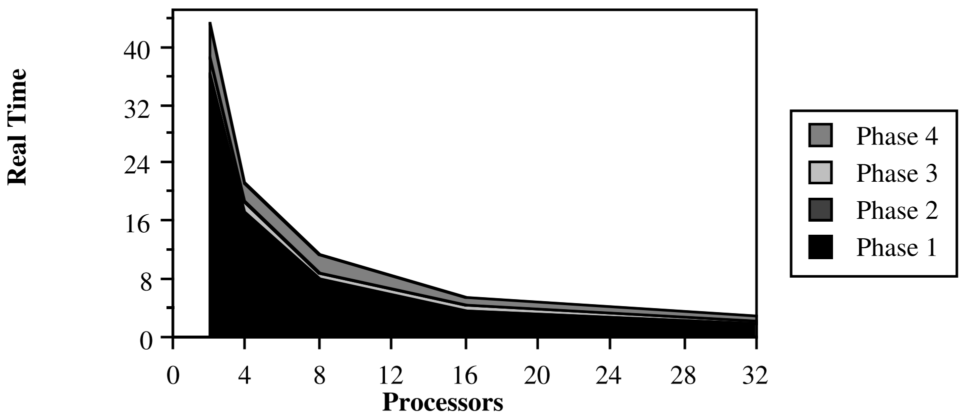



(c) Phase-by-Phase Analysis, percentage of real time

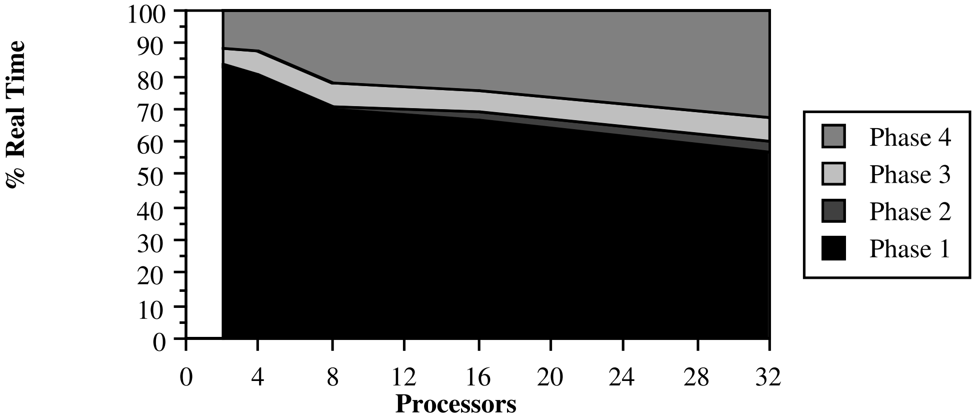
Figure 6: Phase-by-Phase Analysis for BBN TC2000, 800,000 integers




(a) Speedup Curve

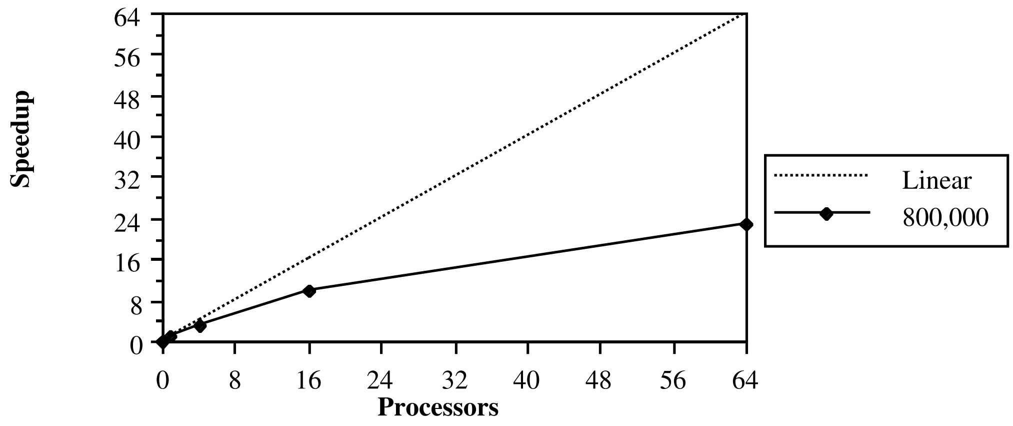



(b) Phase-by-Phase Analysis, real time (seconds)

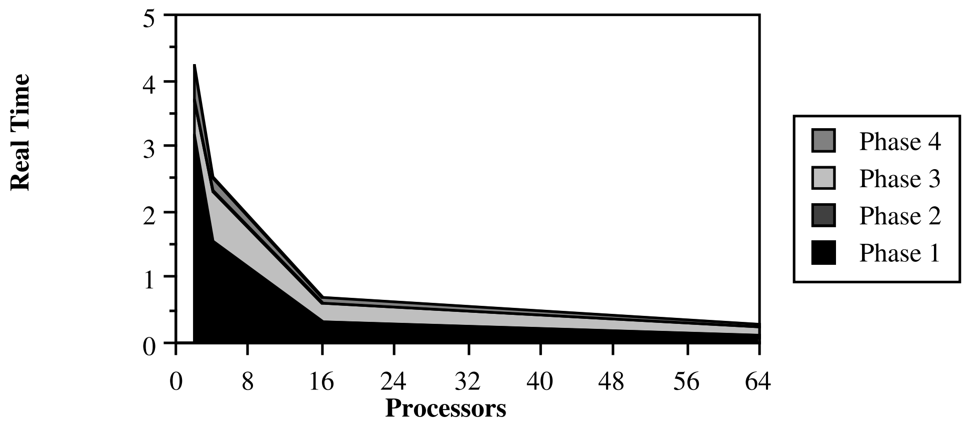



(c) Phase-by-Phase Analysis, percentage of real time

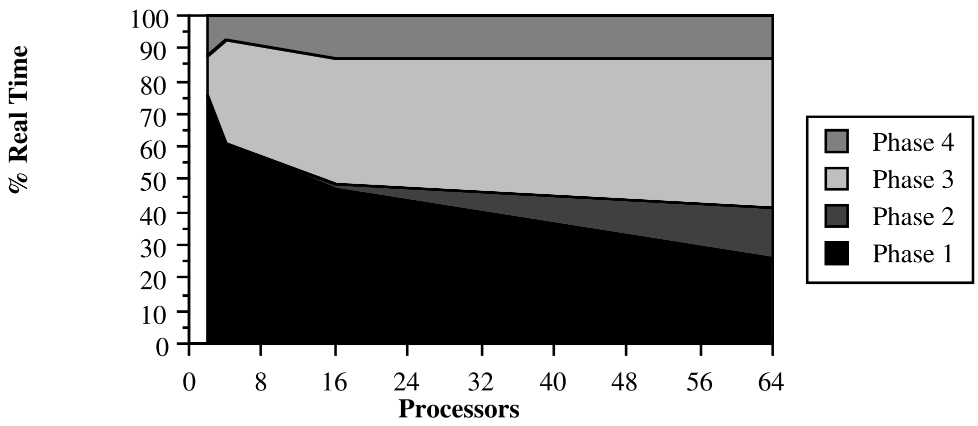
Figure 7: Phase-by-Phase Analysis for iPSC/2-386, 800,000 integers




(a) Speedup Curve

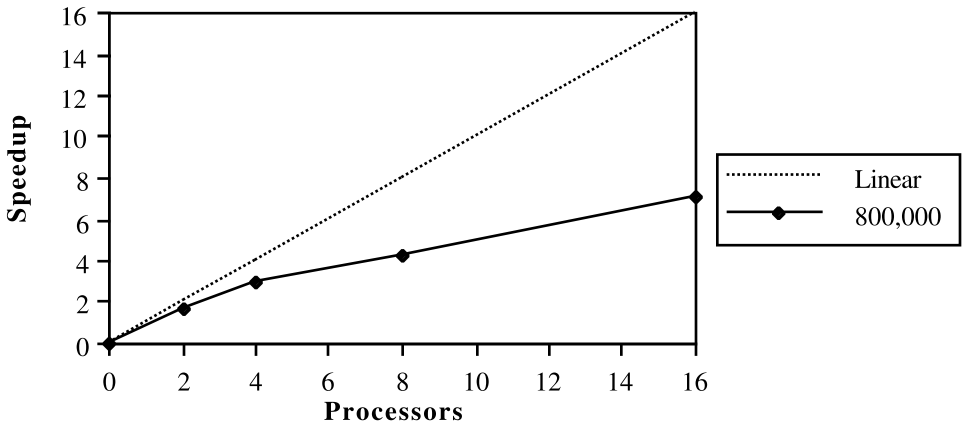



(b) Phase-by-Phase Analysis, real time (seconds)

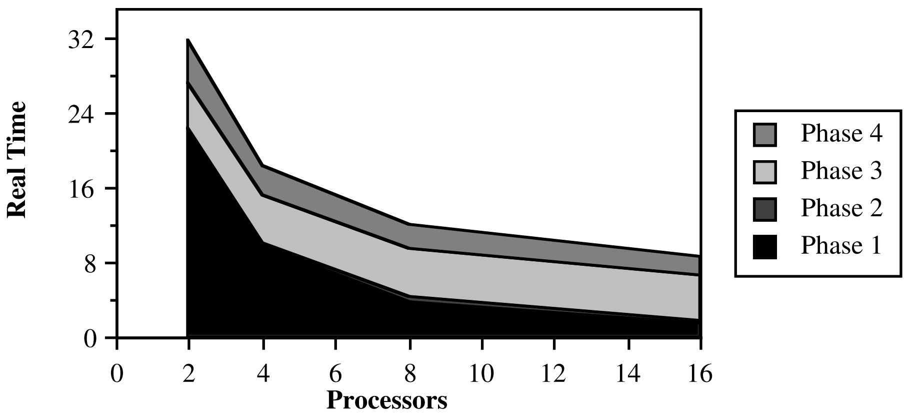



(c) Phase-by-Phase Analysis, percentage of real time

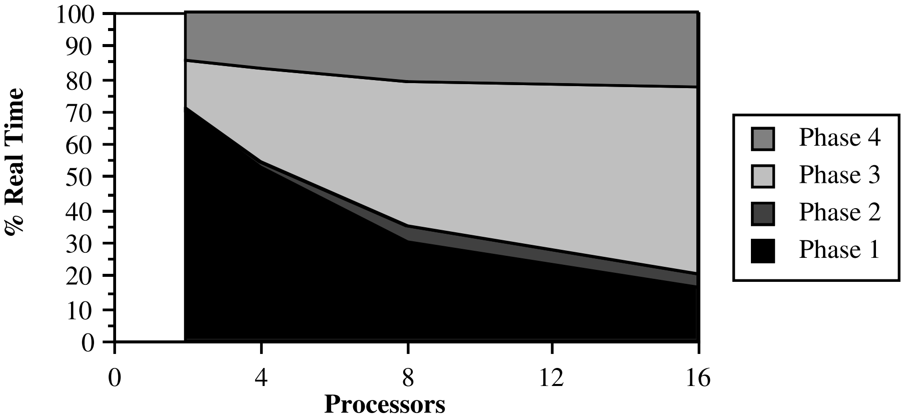
Figure 8: Phase-by-Phase Analysis for iPSC/860, 800,000 integers




(a) Speedup Curve




(b) Phase-by-Phase Analysis, real time (seconds)




(c) Phase-by-Phase Analysis, percentage of real time

Figure 9: Phase-by-Phase Analysis for LAN, 800,000 integers

re-run on the TC2000 sorting 800,000 random integers using 64 processors. The percentage of real time of Phase 2 dropped from 47% (normal sampling) to 24% and the real time for Phase 2 dropped from 0.4s to 0.2s. However, the real time for the entire sort increased from 0.75s with normal sampling to 0.83s with an undersampling factor of 0.5, because the RDFA increased from 1.061 to 1.930. Here, the tradeoff between Phase 2 and Phase 4 was unsuccessful.

In another experiment, using the nonuniform IMOX data described in the next section, sorting 800,000 integers using 64 processors on the TC2000 took 0.75s (RDFA of 1.202), with Phase 2 accounting for 0.35s (47%) of the time. When using an undersampling factor of 0.5, it took 0.67s (RDFA of 1.737), with Phase 2 accounting for 0.19s (28%) of the time. This is merely anecdotal evidence that undersampling can indeed be an effective technique. However, a more thorough study of the cost-benefits of undersampling are beyond the scope of this paper.

Finally, it should be pointed out that while Phase 2 may become a problem when increasing the number processors for a fixed problem size, for a given sorting problem with a fixed number of processors it may not be an issue.

### 4.6 Experiments Using IMOX Data

The previous sections demonstrated that PSRS performs well on random data with an uniform distribution. Most application-generated data, however, does not fit into this simple model. To further illustrate the versatility of PSRS, the algorithm has been tested on a non-random, nonuniform value distribution data set from a real application area. This section reports on the results obtained using the IMOX data set [2], used extensively in image processing and pattern recognition. A pattern matrix, derived from the Munson hand-printed Fortran character set (available from the IEEE Computer Society), consists of 192 binary-coded (24 by 24) handwritten characters from several authors. Each digitized handwritten character is either the letter 'I', 'M', 'O', or 'X', and is represented by an eight-dimensional pattern. The feature numbers are the number of squares from the perimeter of the pattern to the character.

The Euclidean distance for each pair of patterns is calculated. In pattern recognition and clustering analysis these distances need to be sorted, therefore the distance data is used to test PSRS. There are $(192 \times 191)/2 = 18336$ distance values. Examination of the data shows that the it has a bell-shaped distribution curve with most of the numbers clustered around the center of the scale. The data has many duplicates since over 60 values are repeated. For example, one data value is repeated 161 items.

The performance on this data set is influenced by both the high number of duplicates and the nonuniform data. By removing the duplicates, we can see the direct effects of the data distribution on performance. The 18336 numbers have been randomized by adding two more lower-significant digits, resulting in fewer duplicate items. Now there are less than 60 values which are repeated 5 or more times, and no data item is duplicated more than 7 times. The randomization does not change the bell shape of the value distribution curve.

The 18336 values were copied and concatenated many times to form a sufficiently large input to PSRS. Of course, this concatenation process creates an extra duplicate each time the original list is copied. These duplicates are not removed nor randomized any further because many application areas encounter a similar number of ties.

| Sizes | RDFAs | | | | | |
|---|---|---|---|---|---|---|
| | 2PEs | 4PEs | 8PEs | 16PEs | 32PEs | 64PEs |
| 100,000 | 1.015 | 1.044 | 1.032 | 1.198 | 1.154 | - |
| 200,000 | 1.012 | 1.008 | 1.031 | 1.044 | 1.190 | - |
| 400,000 | 1.002 | 1.018 | 1.006 | 1.043 | 1.162 | - |
| 800,000 | 1.000 | 1.001 | 1.007 | 1.009 | 1.075 | 1.202 |
| 1,000,000 | - | 1.001 | 1.003 | 1.022 | 1.028 | 1.089 |
| 2,000,000 | - | - | - | 1.005 | 1.024 | 1.033 |
| 4,000,000 | - | - | - | - | 1.011 | 1.036 |
| 8,000,000 | - | - | - | - | 1.009 | 1.012 |

Table 6: RDFAs for IMOX data, BBN TC2000

The RDFAs are presented in Table 6. There is considerably more variation in the RDFAs with the IMOX data than with the random data. With 64 processors and 800,000 data items, a worst case RDFA of 1.202 was observed. Although these numbers are considerably higher than the uniformly distributed data value case, one should keep in mind that this worst case is still within 20.2% of optimum, a respectable result.

The larger RDFAs for the IMOX data set are due to the effect of a nonuniform data value distribution on pivot selection, and the duplicates present in the data. The pivots selected from the regular sample are heuristic estimates of where to partition the locally sorted data blocks of each processor in order to evenly balance the work of the parallel merge. As heuristics, the pivots are subject to error. In the case of uniformly distributed data values, an error in pivot selection increases the RDFA a fixed amount no matter the location of the pivot in the distribution. In the case of the IMOX's nonuniformly distributed data values, an error in pivot selection may increase the RDFA a great deal if the pivot falls within the peak of the bellcurve.

Despite the adverse effects of the data distribution and the presence of duplicates, the performance of the regular sampling heuristic remains strong, further illustrating the versatility of the algorithm.

# 5 Conclusions

In the field of parallel processing, there are a diverse range of machine architectures. Naturally, the qualitative differences between architectures are exploited to achieve maximum performance. Special architectures demand specialized algorithms, but an efficient algorithm for one class of machine may be inadequate for another class. Consequently, there are a variety of innovative parallel sorting algorithms already in the literature. PSRS is yet another addition to this literature, with its share of weaknesses and strengths.

Its main weakness is the sequential bottleneck of Phase 2, which grows worse as the number of processors increases. Although some solutions have been presented, the lack of experience with PSRS on large configurations of processors makes it impossible to properly assess the problem and its solutions. Clearly, there is a tradeoff between the nature of Phase 2 and Phase 4. Undersampling decreases the size of Phase 2, but results in significantly poorer load balancing in Phase 4. Oversampling increases the size of Phase 2, but improves load balancing in Phase 4. More research is required.

PSRS has significant strengths:

1. **Load Balancing:** The regular sampling heuristic of PSRS is remarkably effective, as shown by both theoretical and empirical analysis. Tight bounds on load balancing are derived and experiments show load balancing to be close to optimal in practice.

2. **Duplicates:** Duplicate keys degrade the performance of load balancing heuristics that inspect the data to estimate the distribution of values. Many parallel sorting algorithms are affected by duplicates, but few researchers address the issue formally. PSRS is relatively insensitive to the presence of duplicates. Specifically, it is proven that the bounds on load balancing change linearly as the number of duplicates increases.

3. **Granularity:** Like all parallel algorithms, having a problem that offers sufficient granularity is mandatory for high performance. PSRS does not achieve good speedups for small problems. Given $n$ keys and $p$ processors, $\frac{n}{p}$ of the work per processor represents the maximum possible granularity. PSRS' approach to the division of work between processors and its effective load balancing allows for the maximum possible granularity. Each processor sorts exactly $\frac{n}{p}$ keys in Phase 1 and, given the near-optimal load balancing, each processor merges approximately $\frac{n}{p}$ keys in Phase 4.

   A different approach is the divide-and-conquer paradigm of some parallel sorting algorithms, as inspired by the recursively divisible geometric hypercube architecture. Basically, the original array of keys is recursively subdivided and then recreated in sorted order. Notably, the granularity of work decreases during the subdivision and some processors are idle various at times, introducing a bottleneck.

   The structure of PSRS, and other similar algorithms ([12, 7], for example), is characterized by a high degree of asynchronous computation and low data movement.

Of the many advantages of the algorithm, perhaps the most important is its suitability for a diverse collection of MIMD architecture classes. The same implementation of the algorithm, modulo the calls to a machine's parallel programming constructs, will achieve good performance on different memory configurations (shared and distributed) and different communications interconnections (LAN, hypercube). While it may not be the best algorithm for any particular architecture, PSRS is a good algorithm for a large class of existing MIMD machines. It is easy to understand and it has excellent theoretical and empirical properties. Given its strengths, PSRS can also be expected to perform well on future MIMD machines.

## Acknowledgments

The financial support of the Canadian Natural Sciences and Engineering Research Council is appreciated, operating grants OGP 8153 and OGP 9198 and a summer scholarship for Paul Lu.

The BBN TC2000 implementation was done using facilities provided by the Massively Parallel Computing Initiative (MPCI) at Lawrence Livermore National Laboratory. Special thanks to Brent Gorda at MPCI for valuable assistance throughout this research. The iPSC/2-386 implementation was done using facilities provided by the Oregon Advanced Computing Institute (OACIS). Thanks to Michael Young of OACIS for his assistance. Access to the iPSC/860 was provided by the Numerical Aerodynamic Simulation (NAS) Division of NASA Ames. Leo Dagum kindly performed the timings for us. Also, thanks to Larry Hall of the Department of Computer Science and Engineering, University of South Florida for access to an iPSC/2-386 early on in this research.

# References

[1] B. Abali, F. Ozguner, and A. Bataineh. Load Balanced Sort on Hypercube Multiprocessors. *5th Distributed Memory Computing Conference*, pages 230–236, 1990.

[2] R.C. Dubes and A.K. Jain. Clustering Techniques: The User's Dilemma. *Pattern Recognition*, 8:247–260, 1976.

[3] K. Birman et al. The ISIS System Manual, Version 2.1. Technical Report TR-91-1185, Computer Science Department, Cornell University, 1991.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, volume 1.* Prentice-Hall, Inc., 1986.

[5] R.S. Francis and I.D. Mathieson. A Benchmark Parallel Sort for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 37(12):1619–1626, 1988.

[6] B.C. Gorda, K.H. Warren, and E.D. Brooks III. Programming in PCP. Technical Report UCRL-MA-107029, Lawrence Livermore National Laboratory, 1988.

[7] J.S. Huang and Y.C. Chow. Parallel Sorting and Data Partitioning by Sampling. *COMPSAC*, pages 627–631, 1983.

[8] E.D. Brooks III. PCP: A Parallel Extension of C that is 99% Fat Free. Technical Report UCRL-99673, Lawrence Livermore National Laboratory, 1988.

[9] Intel. iPSC/2 Programmer's Reference Manual. Technical report, Intel Scientific Computers, Beaverton, Oregon, 1989.

[10] P.P. Li and Y.W. Tung. Parallel Sorting on the Symult 2010. *5th Distributed Memory Conference*, pages 224–229, 1990.

[11] T.A. Marsland, T. Breitkreutz, and S. Sutphen. A Network Multi-processor for Experiments in Parallelism. *Concurrency: Practice and Experience*, 3(3):203–219, 1991.

[12] M.J. Quinn. Parallel Sorting Algorithms for Tightly Coupled Multiprocessors. *Parallel Computing*, 6:349–357, 1988.

[13] D. Rotem, N. Santoro, and J. Sidney. Distributed Sorting. *IEEE Transactions on Computers*, 34(4):372–376, 1985.

[14] S. Seidel and L.R. Ziegler. Sorting on Hypercubes. In M.T. Heath, editor, *Hypercube Multiprocessors*, pages 285–291. SIAM, 1987.

[15] H. Shi. Parallel Sorting on Multiprocessor Computers. Master's thesis, University of Alberta, 1990.

[16] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[17] T. Tang. Parallel Sorting on the Hypercube Concurrent Processor. *5th Distributed Memory Conference*, pages 237–240, 1990.

[18] P.J. Varman and K. Doshi. Sorting with Linear Speedup on a Pipelined Hypercube. *IEEE Transactions on Computers*, 41(1):97–103, 1992.

[19] B. Wagar. Hyperquicksort: A Fast Sorting Algorithm for Hypercubes. In M.T. Heath, editor, *Hypercube Multiprocessors*, pages 292–299. SIAM, 1987.

[20] L.M. Wegner. Sorting a Distributed File in a Network. *Computer Networks*, 8:451–461, 1984.

[21] M. Wheat and D.J. Evans. An Efficient Parallel Sorting Algorithm For Shared Memory Multiprocessors. *Parallel Computing*, 18:91–102, 1992.

[22] Y. Won and S. Sahni. A Balanced Bin Sort for the Hypercube Multicomputers. *Journal of Supercomputing*, 2:435–448, 1988.