The Bitonic Sort on Transputer Architectures

Tim Oliver
Centre for Mathematical Software Research
University of Liverpool

September 1991

Abstract

The bitonic sort algorithm is a parallel sorting algorithm that has been implemented in sorting networks and is readily adaptable to Transputer arrays. This paper looks at the implementation and time cost of the algorithm for a $\mathcal{PUMA}$ machine and compares this with an implementation on T8 networks to see the benefits that the new architecture provides over the previous generation of Transputers.

1 Introduction

The bitonic sort algorithm is not the most efficient sequential sorting algorithm having a cost proportional to:

$$\tfrac{1}{4} n \log_2 n (\log_2 n + 1).$$

However, the algorithm is especially designed for execution by parallel processing elements. Initially the algorithm was implemented on sorting networks [1] but a parallel form of the algorithm is also suitable for execution on local memory MIMD architectures such as Transputer arrays. Previous work under Esprit Supernode project P1085 developed an implementation of the algorithm for a T8 network [4]; however this algorithm had a poor performance because of the large communication overhead involved in exchanging data between distant processors at each stage of the algorithm. The point to point communications between any pair of processors that the $\mathcal{PUMA}$ architecture provides greatly reduces the communication burden on the processors and so should lead to a more efficient parallel algorithm. This paper presents a parallel bitonic sort algorithm for the $\mathcal{PUMA}$ architecture and compares its performance with that of a Supernode algorithm.

To predict the performance of this algorithm on a $\mathcal{PUMA}$ machine we develop a simple cost model. The cost model expresses the execution time of the algorithm in terms of a set of hardware constants and a set of problem parameters.

The hardware constants measure the time taken to compare and communicate elements of the vector to be sorted. Although the algorithm will sort elements of any basic type, e.g., `INT` or `REAL32`, each type will have a different set of values for the hardware constants. For example, comparison of two `INT` values may have a different cost from comparing two `REAL32` values, and the communication of a `REAL64` costs more than the communication of a `REAL32`. So, when presenting the cost of the algorithm for particular problems we have chosen to use hardware constants for the `REAL32` data type.

We define three hardware constants:

T_e the time taken to compare two `REAL32` elements. $T_e = 100ns$.

T_s,T_c the total time taken to communicate a message of n `REAL32`s is given by $T_s + nT_c$. $T_s = 1000ns$, $T_c = 880ns$.

The comparison cost simply assumes an average rate of 10 MFlops for numerical operations. Communication rates are more difficult to model and are discussed in [5]. For this paper we assume that the underlying switch network is a three stage cross-bar network as recommended by Hofestädt *et al* [3]. The values

for T_s and T_c are derived from [5] assuming a single channel, long message communications model. For simplicity, we permit all components of the communication cost to be overlapped with communication on other links and comparison operations; for example we assume that transmitting a vector on one link has the same time cost as transmitting that complete vector on each of the 4 links in parallel. This assumes that the communication engines and memory unit can satisfy these demands. The values of these constants are only approximate; a long message could be partitioned into smaller messages and these transmitted in parallel on multiple channels making better use of the link bandwidth.

In addition to these hardware constants two problem parameters are needed:

n the problem size; in this case the number of elements in the vector to be sorted,

p the number of processors the algorithm will use.

Presented in its simplest form, the algorithm requires both n and p to have values that are powers of two, i.e., $n = 2^k$, $p = 2^m$, $(m \leq k)$. These constraints on n and p are assumed throughout the rest of this paper.

The following section describes the generic bitonic sort algorithm for MIMD architectures (Section 2). The $\mathcal{PUMA}$ algorithm is described and modelled in Section 3. Section 4 compares the performance of the $\mathcal{PUMA}$ algorithm with that developed for a T8 machine. Finally, our conclusions are presented briefly in Section 5.

2 The algorithm

A bitonic sequence of n $(= 2^k)$ elements $\{x_0, x_1, \ldots, x_{n-1}\}$ is such that

$$x_0 \leq x_1 \leq \ldots \leq x_{j-1} \leq x_j \geq x_{j+1} \geq \ldots \geq x_{n-2} \geq x_{n-1}$$

or the sequence can be shifted cyclically to satisfy this condition.

At the heart of the bitonic sort algorithm is a routine which merges a bitonic sequence to give a sorted sequence. To merge a bitonic sequence of n elements:

Algorithm 1 (Bitonic merge)

1 let $i = n$

2 while $i \geq 2$

2.1 for each element x_j in the sequences of elements

$$x_0 \ldots x_{i/2-1}, \quad x_i \ldots x_{3i/2-1}, \quad \ldots, \quad x_{n-i} \ldots x_{n-i/2-1}$$

compare each element x_j with the element $x_{j+i/2}$;
for ascending order, if $x_j > x_{j+i/2}$ then exchange elements,
for descending order, if $x_j \leq x_{j+i/2}$ then exchange elements

2.2 let $i = i/2$

□

To construct a sorted sequence from an unsorted sequence of length n the bitonic merge procedure is used to produce successively larger sequences of sorted elements:

Algorithm 2 (Bitonic sort)

1 let i = 2

2 while $i \leq n$

2.1 for each bitonic sequence of elements

$$x_0 \ldots x_{i-1}, \quad x_i \ldots x_{2i-1}, \quad \ldots, \quad x_{n-i} \ldots x_{n-1}$$

sort the sequence using bitonic merging with alternating sort order.
For ascending sort order, sort first sequence into ascending order and second into descending order, e.t.c., and *vice versa* for descending sort order.

2.2 let $i = 2i$

□

The adaptation of this algorithm to sort n elements on a MIMD machine with p ($= 2^m$, $m \leq k$) processors is as follows:

Initially the root process distributes the unsorted vector blockwise to the processes, keeping a block itself, with each process receiving n/p elements. To build up a sorted sequence on each process a sequential quicksort routine is used as it has a better performance than sequential bitonic sorting. Alternate processes perform sorting in ascending and descending order. Now each pair of adjacent processes holds a bitonic sequence.

Next the merging stage is entered as followed:

Algorithm 3 (Parallel bitonic sort)

1 let $j = 2$

2 while $j \leq p$

2.1 let $i = j$

2.2 while $i > 1$

2.2.1 for each process s_k in the blocks of processes

$$s_0 \ldots s_{i/2-1}, \quad s_i \ldots s_{3i/2-1}, \quad \ldots, \quad s_{p-i} \ldots s_{p-i/2-1}$$

processes s_k and $s_{k+i/2}$ perform a compare-exchange operation on their vector elements: element $s_k[ii]$ (the element at index ii of the vector on process s_k) is compared with element $s_{k+i/2}[ii]$.
The details of this step are determined by the architecture; see Section 3 for details.

2.2.2 let $i = i/2$

2.3 perform bitonic merging (Algorithm 1) on each process

2.4 let $j = 2j$

□

The complete sequence has now been sorted and each process passes its vector back to the root process.

Step 2.2.1 of the parallel algorithm contains all the communications for the algorithm except for the initial distribution and final gather of the vector. Hence, the detailed code for this step will be optimised to reduce or hide the communication cost as much as possible for each architecture. This results in different implementation details for the Supernode and $\mathcal{PUMA}$ algorithm.

3 $\mathcal{PUMA}$ algorithm

A first approach to Step 2.2.1 is for each pair of processes s_k and $s_{k+i/2}$ to exchange half of their vectors: process s_k sends its high half vector to process $s_{k+i/2}$ and receives the low half vector of $s_{k+i/2}$ in exchange. These two communications can be executed in parallel provided the communicated vectors are input into temporary vectors of size $n/2p$. Process s_k then performs a compare-exchange operation on elements from its low half vector and the low half vector received from $s_{k+i/2}$. Similarly, process

$s_{k+i/2}$ performs a compare-exchange operation on the high half vectors it holds. The step is completed by returning the high half vector to s_k and the low half vector to $s_{k+i/2}$ in parallel. So this method has 2 communications operations per step, each operation consisting of the communication of 2 half vectors in parallel.

This method keeps all the processes busy, but the total amount of communication performed is greater than necessary: the half vectors are returned to their source at the end of the iteration and then at the beginning of the next iteration output again if the source process has the same state as in the previous iteration (either expecting high or low half vectors). The communication cost can be reduced by fetching the half vectors at the beginning of the step directly from their location in the previous iteration. This removes the need to return half vectors at the end of the step, but may instead require two half vectors of a process to be input at the start of the step: both the half vector it would input under the previous method and its own half vector for this iteration if this was stored on a different process in the previous iteration. The total amount of communication in the worst case is as much as for the previous method, i.e., 4 half vectors per iteration, but all 4 communications can be executed in parallel in one operation at no additional cost in storage space, compared with 2 operations for the previous method.

Simple expressions specify the location of the 2 half vectors that process s_k requires in iteration i of Algorithm 3. We define the state of process s_k, $oddk$, as

$$oddk = (k \bmod i/2) \operatorname{rem} 2.$$

If $oddk$, i.e., $oddk = 1$, then process s_k requires high half vectors, otherwise it requires low half vectors. Process s_k is paired with process s_{k1} where

$$k1 = k + i/2 - (i * oddk).$$

In the previous iteration the state of process s_{k1} was

$$oddk1p = (k1 \bmod i) \operatorname{rem} 2,$$

and it was paired with process s_{k2}

$$k2 = k1 + i - (2i * oddk1p).$$

If $oddk1p$ then s_{k1} holds high half vectors from the previous iteration and s_{k2} holds low half vectors. Hence, from these expressions each process can calculate whether it needs to input vectors from s_{k1} or s_{k2} and whether these vectors are stored in low or high half vectors on those processes.

To find its own half vector, similar expressions are used. In the previous iteration s_k had state

$$oddkp = (k \bmod i) \operatorname{rem} 2,$$

and was paired with process s_{k3} where

$$k3 = k + i - (2i * oddkp).$$

If $oddkp$ then s_k holds high half vectors from the previous iteration and s_{k3} holds low half vectors. Hence, from these expressions each process can calculate whether it needs to input its own high or low half vector from process s_{k3}.

These expressions are more clearly shown in Figure 1 as two tables which give the location of the 2 half vectors that process s_k requires at iteration i for the various values of $oddk$, $oddkp$ and $oddk1p$.

Similar expressions can be found for the destination processes for the half vectors held by a process at the start of an iteration.

This operation is repeated for the loop in Step 2.2 of the algorithm. On exit from this loop the half vectors must be returned to their correct processes before the next step.

This second method decreases the communication cost of the algorithm by performing communications in parallel, however, even better performance can be achieved by overlapping communications with comparisons. Step 2.2 may be rearranged to use parallel threads which operate on quarter vectors, i.e., halves of the half vectors. Whilst one thread performs the compare-exchange on a pair of quarter vectors, another thread inputs the next pair of quarter vectors. Step 2.2 becomes:

Figure 1: Locations of half vectors

Algorithm 4 ($\mathcal{PUMA}$ main loop)

2.2.1 process s_k gets first $1/4$ vector from $s_{k+i/2}$

2.2.2 while $i > 1$

2.2.3 par
perform compare-exchange on first $1/4$ vector
get second $1/4$ vector

2.2.4 par
perform compare-exchange on second $1/4$ vector
get first $1/4$ vector for next iteration (if $i > 2$)

2.2.5 let $i = i/2$

2.2.6 return $1/4$ vectors to owner processes □

This third method requires the same amount of storage as the previous method, but hides most of the communications behind compare-exchange operations.

The cost model for this last algorithm on a $\mathcal{PUMA}$ machine is derived as follows:

If the vector to be sorted is stored on one root process then we must include the cost of the initial distribution of the unsorted vector, and the collection of the sorted vector back to the root at the end of the algorithm. The algorithm and cost model for these standard operations is given in [6]. They have a combined cost of

$$\frac{p-1}{2}\left(T_s + \frac{n}{p}T_c\right).$$

If the vector is already distributed before the algorithm starts we leave the sorted vector distributed also.

The main algorithm begins with each process performing an initial quicksort on its vector. This costs

$$\frac{n}{2p}\log_2\frac{n}{p}T_e.$$

The parallel threads in Step 2.2 cost

$$\max\left(T_s + \frac{n}{4p}T_c, \frac{n}{4p}T_e\right).$$

The loop in i executes $\log_2 j$ times, and on the last iteration the second thread only performs a compare-exchange operation. Before the first iteration a $1/4$ vector is input at cost $T_s + (n/4p)T_c$. At the end of the loop two $1/4$ vectors are returned to their owner process in parallel at cost $T_s + (n/4p)T_c$. This gives a total cost for the i loop of:

$$(2\log_2 j - 1)\max\left(T_s + \frac{n}{4p}T_c, \frac{n}{4p}T_e\right) + 2(T_s + \frac{n}{4p}T_c) + \frac{n}{4p}T_e.$$

After this loop ends each process performs a bitonic merge (Algorithm 1) on its own vector at cost

$$\frac{n}{2p}\log_2\frac{n}{p}T_e.$$

All of this work in the loop of Step 2 is repeated over j, with $j = 2^k$ where $k = 1 \ldots \log_2 p$. The total cost of this step is thus

$$\max\left(T_s + \frac{n}{4p}T_c, \frac{n}{4p}T_e\right)(\log_2 p)^2 + 2(T_s + \frac{n}{4p}T_c)\log_2 p + \left(1 + 2\log_2 \frac{n}{p}\right)\frac{n}{4p}\log_2 pT_e.$$

This gives the total cost for a pre-distributed bitonic sort as

$$\begin{aligned}&\tfrac{n}{2p}\log_2 \tfrac{n}{p}T_e + \max\left(T_s + \tfrac{n}{4p}T_c, \tfrac{n}{4p}T_e\right)(\log_2 p)^2 + \\ &2\left(T_s + \tfrac{n}{4p}T_c\right)\log_2 p + \left(1 + 2\log_2 \tfrac{n}{p}\right)\tfrac{n}{4p}\log_2 pT_e.\end{aligned}$$

The total cost for a parallel bitonic sort of a vector initially on a single processor is:

$$\begin{aligned}&\tfrac{p-1}{2}\left(T_s + \tfrac{m}{p}T_c\right) + \tfrac{n}{2p}\log_2 \tfrac{n}{p}T_e + \max\left(T_s + \tfrac{n}{4p}T_c, \tfrac{n}{4p}T_e\right)(\log_2 p)^2 + \\ &2\left(T_s + \tfrac{n}{4p}T_c\right)\log_2 p + \left(1 + 2\log_2 \tfrac{n}{p}\right)\tfrac{n}{4p}\log_2 pT_e.\end{aligned}$$

4 Performance

The performance of the parallel algorithm is presented graphically in terms of the algorithm's speedup over quicksort, the best sequential algorithm. The speedup shows how many times faster the parallel algorithm is compared with the sequential algorithm. For a discussion of speedup and related performance issues see Section 1.3, Characterisation of Performance, in [2]. A range of problem sizes from 1MWords up to 256 MWords is used to show the variation in performance with problem size. These are big problems; 1MWord is a vector of 10^6 REAL32 elements. The speedup is also plotted over a wide range in the number of processors to indicate how well an algorithm would perform on large arrays of processors.

Figure 2 shows the speedup of the $\mathcal{PUMA}$ algorithm that is predicted by the cost model detailed in the previous section. Two graphs are shown: the top graph shows the speedup predicted when the vector is pre-distributed across the processors, the bottom graph shows the speedup when the vector initially resides on a single source processor. The first graph shows that the pre-distributed algorithm scales very well for large networks of processors. The actual speedups achieved are very low however. There are two reasons for this. The main reason for the poor speedup of the algorithm is that the total number of operations performed by the bitonic algorithm to sort a vector, i.e., the sequential bitonic sort cost, is much greater than the cost of sequential quicksort. Hence, even if the parallel algorithm was perfectly efficient, the speedup achieved over quicksort would be much lower than the number of processors. The bitonic sort algorithm remains popular despite this drawback because it has a natural parallel expression whereas the quicksort algorithm is very difficult to parallelize. The second reason for the low speedup is that the parallel algorithm is not very efficient on the T9 architecture. This is because of the ratio between the comparison time and the communication time. In the main loop of the $\mathcal{PUMA}$ algorithm (see Algorithm 4) both parallel threads operate on quarter vectors. The compare-exchange thread performs $n/4p$ operations whilst the communication thread inputs and outputs in parallel $n/4p$ elements. With the current estimates for T_e, T_c and T_s the communication thread has a much greater cost and so for a lot of the time the arithmetic units are idle until communication finishes. If the cost of a compare-exchange operation was more than the cost to communicate an element then almost all communication would be hidden behind comparisons and the algorithm would show very good efficiency. For REAL32 elements this is unlikely for any architecture since communication between processors will always take longer than local memory accesses and the comparison itself has a very low cost. On the other hand, sorting string elements in a database would have a better balance between communication and comparison cost.

The second graph showing the speedup for the single source sort has a much lower performance than that for the pre-distributed sort shown in the first graph. The difference in cost between these two situations is the time taken to distribute the vector at the start of the algorithm and collect the sorted vector back at the end. The single source processor becomes a bottleneck as it communicates data to every other processor. As the number of processors is increased, the time taken to sort the distributed vector decreases. However,

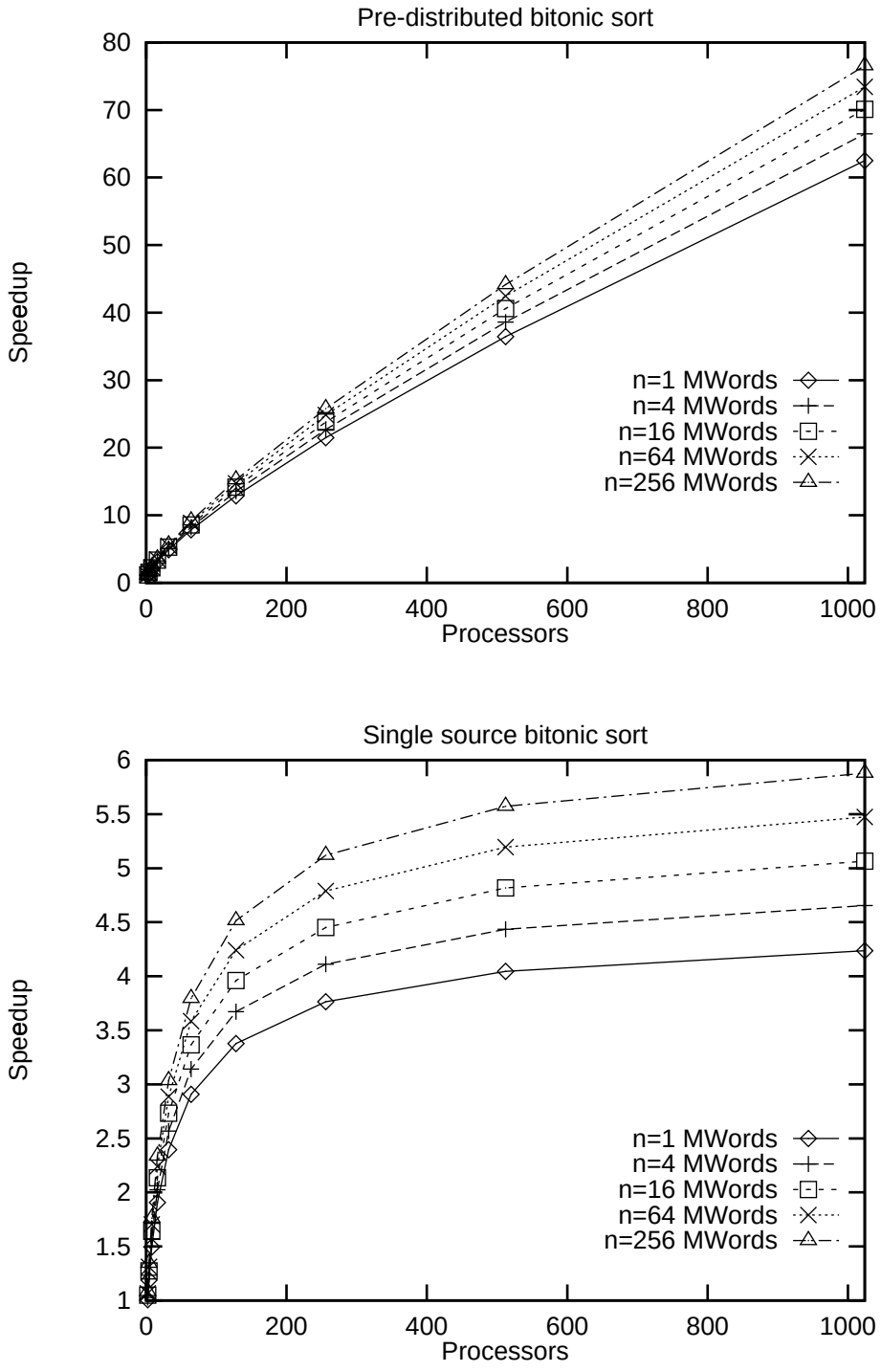

Figure 2: Performance of bitonic sort on the T9 architecture

the cost to distribute vector segments from the single source to an increasing number of processors increases rapidly. The large distribution cost results in much smaller speedups for the single source sort across the range of machine sizes. This cost also changes at a greater rate than the pre-distributed bitonic sort cost as the number of processors increases. This leads to a reduction in the speedup change as the number of processors increases as is shown by the plateau in the graph.

As a comparison, the performance of the bitonic sort algorithm on a T8 machine is shown in Figure 3. Again, the top graph shows the cost for a pre-distributed bitonic sort and the bottom graph the cost for a single source sort. The speedup is that achieved over quicksort running on a single T8 processor. Full details of the parallel algorithm and cost model are given in [4], however, a few comments are useful. The original T8 algorithm uses a chain of processors and packetizes vector communications to improve the performance of communication between distant processors. The algorithm does not however overlap comparisons and communication. The T8 processor also has different comparison and communication costs from the T9. This means that the graphs cannot be used to compare the actual run-time costs of the algorithms on the different architectures. However, the T9 is several times faster than the T8 processor and so the T9 algorithm gives better run-time costs than the T8 algorithm.

The first graph in Figure 3 shows very poor speedup over sequential quicksort. This is because of the high cost of the communications. A more important feature to notice is that the speedup reaches a plateau whereas for the T9 algorithm the speedup increases continually as the number of processors is increased. This plateau is due to the need to buffer individual communications between distant processors through increasing numbers of intermediate processors as the total network size increases. With the T9 architecture the cost of each individual communication does not increase as the total number of processors increases. The speedup of the single source algorithm, which is shown in the second graph, is not much worse than the pre-distributed speedup. This indicates that the packetized communications used to distribute the unsorted vector do not add as significant a cost to the algorithm as was the case for the T9 algorithm.

Comparing the two figures for the T9 and T8 architectures shows that the pre-distributed algorithm has benefited greatly from the direct communications between all processors in the T9 architecture. However, the performance of the T9 single source algorithm is crippled by the distribution and collection cost of the full vector. This cost cannot be reduced because of the limited total bandwidth of the 4 links on the source processor. The affect of the distribution is so great because the pre-distributed algorithm cost itself is relatively low. Pre-distributed parallel algorithms in other fields of numerical methods often have a much higher cost which means that the distribution cost is much less significant and both forms of the algorithm will give a similar performance. The total cost for all pre-distributed parallel sort algorithms is low and so this distribution bottleneck will affect all the sort algorithms and result in poor speedups.

The low efficiency of the T9 parallel algorithm can be improved somewhat by the use of a variation in the bitonic algorithm. Instead of performing compare-exchange operations on the vector pairs, this improved algorithm merges and splits the sorted vectors. The communication pattern remains the same as for the original algorithm. This change doubles the arithmetic cost in the parallel threads and removes the need for the bitonic sort on each processor at the end of each loop. Hence, given the current ratio between communication and comparison the extra work performed in the threads is still hidden behind communications and the overall cost of the algorithm is reduced. Unfortunately, we did not have time to study the cost of this algorithm in any detail.

5 Conclusions

The bitonic sort algorithm maps naturally onto local memory, MIMD machines such as $\mathcal{PUMA}$. The pre-distributed parallel algorithm gives reasonable performance and scales very well onto large T9 networks. This is due to the ability to communicate directly between all T9 processors in the network. Unfortunately, the relatively high cost of initial distribution means that the single source algorithm has a very poor performance. In conclusion, the parallel bitonic sort algorithm can be recommended for applications where the data is pre-distributed, especially for extremely large vectors which cannot be held on a single processor. For smaller vectors which are initially stored on a single processor, a sequential quicksort is easier to use and is only a few times slower than the single source parallel algorithm. Other parallel sort algorithms may yield a greater efficiency than the bitonic sort, but all will give only a poor performance on the T9 when

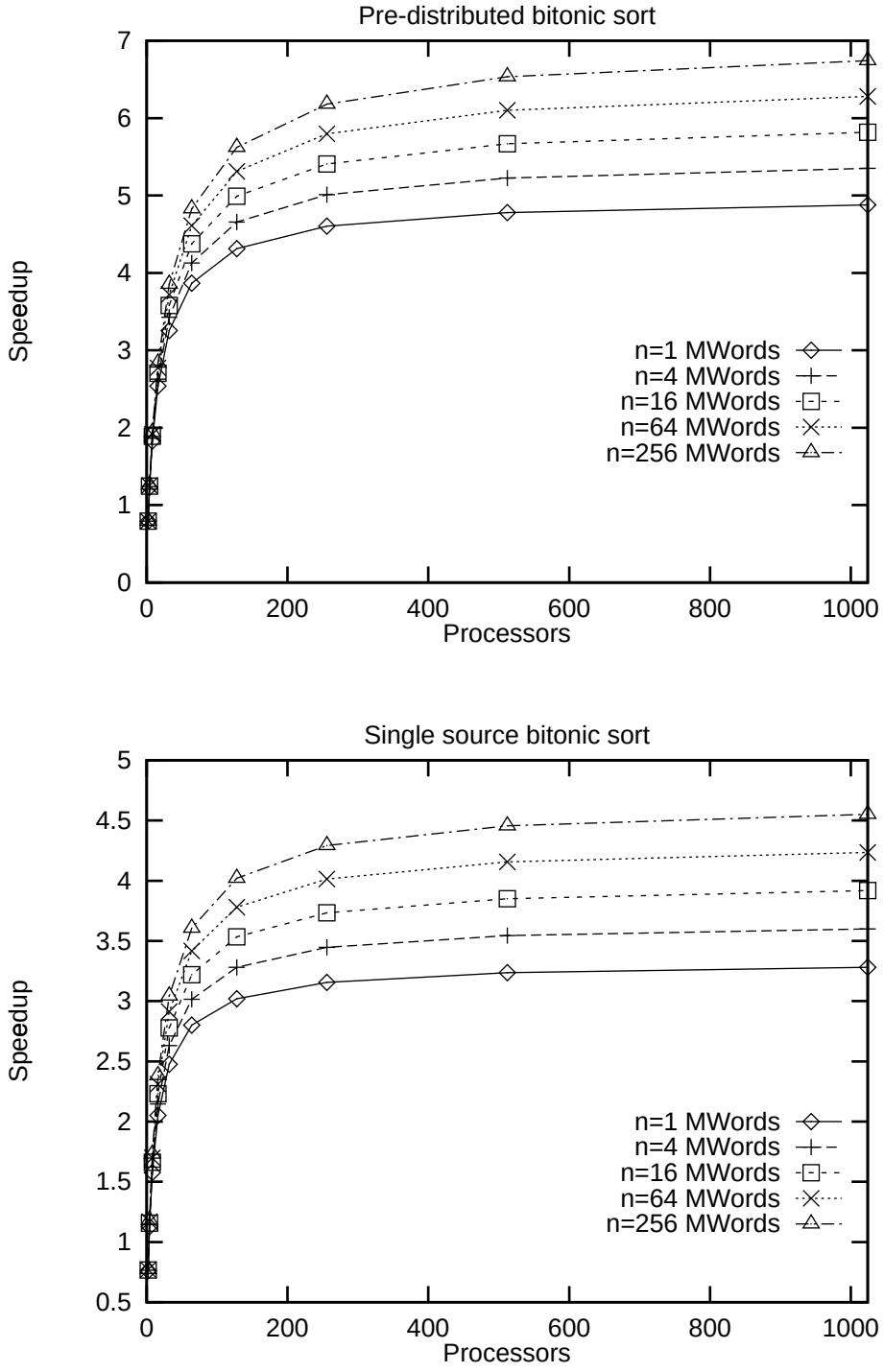

Figure 3: Performance of bitonic sort on the T8 architecture

initial distribution is required.

References

[1] S. G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[2] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: architecture, programming and algorithms*. Adam Hilger, 1988.

[3] Holm Hofestädt, Axel Klein, and Erwin Reyzl. Investigation of dynamically switched, scalable network structures. PUMA Deliverable 2.1.1, Siemens AG, October 1990.

[4] Tim Oliver. Sorting algorithms for transputer arrays. Working paper, Centre for Mathematical Software Research, University of Liverpool, October 1988.

[5] Tim Oliver. A communications model for a PUMA machine. PUMA Working paper 17, University of Liverpool, October 1990.

[6] Tim Oliver. Parallel algorithms for the BFGS update on a PUMA machine. PUMA Working paper 32, University of Liverpool, September 1991.