

# A Practical Algorithm for Integer Sorting on a Mesh-Connected Computer \*

Nathan Folwell

Sumanta Guha †

Ichiro Suzuki ‡

Department of Electrical Engineering  
and Computer Science  
University of Wisconsin-Milwaukee  
P.O. Box 784  
Milwaukee, WI 53201  
Fax: (414) 229-6958  
Email: [folwell, guha, suzuki]@cs.uwm.edu

## Abstract

This paper presents count-sort, a parallel algorithm for mesh-connected computers to sort integers where the range of inputs is known. A straightforward counting technique that has not been implemented previously in parallel sorting algorithms is presented. On a mesh-connected computer with  $\sqrt{N} \times \sqrt{N}$  processors we are able to sort  $N$  integers in the range  $1 \dots \sqrt{N}$  in time  $c\sqrt{N}$  where  $c$  is very small. For practical values of  $N$ , the algorithm is extremely fast. Further, it is possible to expand the range by a factor  $k$  to  $1 \dots k\sqrt{N}$  so that the slowdown is less than  $k$ .

We produce two implementations of count-sort on the SIMD MasPar MP-1 with 8192 processors. The first sorts 8-bit integers, one per processor, significantly faster than the manufacturer's current library routine for sorting 8-bit integers. The second implementation is a fast version that sorts several elements per processor.

*Keywords:* Parallel algorithms, Sorting, Mesh-connected computer.

## 1 Introduction

The study of parallel algorithms is increasingly becoming one of the most important areas in computer science. A very practical and interesting architecture for parallel algorithms is the mesh. Its regular interconnection, ideal for VLSI implementation, is easily scalable.

A fundamentally important problem for the mesh-connected architecture is that of finding efficient sorting algorithms. In fact, sorting is often a key step in other mesh algorithms. Several practical  $O(\sqrt{N})$  time algorithms to sort on a  $\sqrt{N} \times \sqrt{N}$  mesh have been proposed [4, 6, 7, 11]. In the model where there is initially one element per processor and the target order is snake-like row-major, Schnorr and Shamir [10] developed an algorithm that runs in time  $3\sqrt{N} + o(\sqrt{N})$ , which is asymptotically near optimal as a provable lower bound is  $3\sqrt{N} - o(\sqrt{N})$  [5, 10]. However, their algorithm is only practical for very large  $N$ . More recently, Krizanc [3] presented the first deterministic sorting algorithm in a similar model that overcomes the  $3\sqrt{N} - o(\sqrt{N})$  bound *given* that input is drawn from integers in the

---

\*A preliminary version of this paper appears in *Proceedings of the High Performance Computing Symposium'95*.

†Communicating author.

‡Supported in part by the National Science Foundation under grants CCR-9004346 and IRI-9307506, and the Office of Naval Research under grant N00014-94-1-0284.

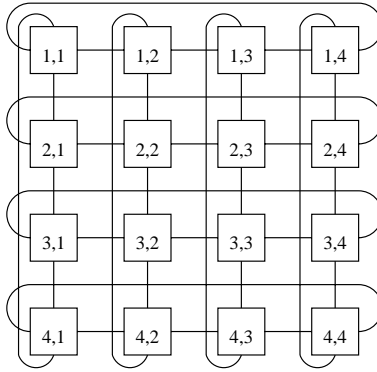


Figure 1: A  $4 \times 4$  mesh with wrap-around connections.

range  $1 \dots N$ , by using counting techniques. This is analogous to the situation in the sequential model where, given information about the range of inputs, it is possible to sort faster than the lower bound of  $\Omega(N \log N)$  that holds for arbitrary inputs [2].

We present a parallel sorting algorithm, *count-sort*, for mesh-connected computers that sorts  $N$  integers in the range  $1 \dots k\sqrt{N}$  faster than the above algorithms for practical values of  $k$  and  $N$ . Count-sort is fast because it is *not* comparison based. Instead, a counting technique is used to achieve high speeds. Further, it is practical, and we have, in fact, implemented it as an extremely fast sorting routine on the MasPar MP-1: on an 8192 processor MasPar MP-1 our routine sorts 8-bit numbers 30% faster than the current 8-bit sorting routine in the MasPar software library. Such routines to sort “short” integers have many applications.

In Section 2 we define our models of computation. In Section 3 we describe count-sort and analyze its running time. We first develop the algorithm on a simple model of computation. Next, we modify the algorithm for a more powerful model that better resembles machines currently available on the market. In Section 4 we present implementations of count-sort on the commercially available MasPar MP-1, with time comparisons between our implementations and other available sorting implementations. Section 5 presents conclusions and possible extensions to our algorithm.

## 2 Models of Computation

Here we present two models of computation for analyzing our algorithm. The first is a simple model to develop the algorithm, while the second has additional capabilities.

### 2.1 Simple Model of computation

Assume there are  $N$  processors which are arranged in a  $\sqrt{N} \times \sqrt{N}$  mesh. Each processor is connected to its four nearest neighbors. Processors on the perimeter of the mesh have wrap-around connections. We identify each processor with a unique ID of the form  $(i, j)$  where  $i$  is the row number and  $j$  is the column number (see Figure 1). These ID numbers can also be used to identify processors in row major order. For example, in Figure 1, the processor with ID  $(2, 3)$  is the 7th processor in row major order for a  $4 \times 4$  mesh.

Each processor can perform simple programming operations and route a single value to one of its four neighbors in constant time.

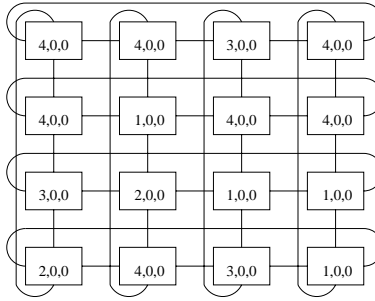


Figure 2: Initial configuration of *scratch*, *count*, and *output* shown in order.

The programming operations that are performed are similar to any high level programming language: the conditional **if** statement, the assignment statement, logical and arithmetic operations are all assumed to execute in time  $t_O$ .

Define the route command to be  $\text{SEND}_{\{N,S,E,W\}}[var]$ . For example,  $\text{SEND}_N[input]$  would send *input* on every processor to the *input* register to the north. This includes the wrap-around connections. Assume the SEND operation executes in time  $t_S$ .

All operations are performed simultaneously in SIMD manner on all processors unless specified by a conditional statement. If a conditional statement is used then the processors where the conditional is true will perform the operation while the other processors are idle.

## 2.2 A More Powerful Model

It is useful to analyze our algorithm on a more powerful model that better represents machines currently available on the market. This model has three additional capabilities.

The first capability which is available, for example, on the MasPar MP-1, allows for *full permutation routing* in constant time. Define this operation to be  $\text{PERMUTE}[var,dest]$ , which routes the values in *var* to the processors with ID value *dest*. Note that *dest* is a variable on each processor. This is a powerful operation. It is implemented on the MP-1 with a three stage hierarchy of crossbar switches, called the router [1, 12]. The time for this operation is  $t_P$ .

The second capability, also available on the MasPar MP-1, allows a variable to be sent in any of the four compass directions an arbitrary number of steps in constant time provided the intermediate processors are idle. Define this operation to be  $\text{SEND}[dist]\{N,S,E,W\}[var]$ . For example, in Figure 1,  $\text{SEND}[3]S[input]$  sends the contents of *input* from row 1 to row 4 in constant time if processors in rows 2 and 3 are idle. The time for this operation is  $t_{SD}$ .

The third capability is  $\text{SEND\_COPY}$ , which is the same as the more powerful SEND, but a copy of *var* is left in processors along its path. For example,  $\text{SEND\_COPY}[3]S[input]$  still sends *input* three processors to the south, but each intermediate *input* register gets a copy of the original *input* as well. The time for this operation is  $t_C$ .

These three capabilities are common not only to the MP-1. Other commercial machines such as the MasPar MP-2, Cambridge Parallel Processing DAP, and the DEC MPP have similar capabilities.

## 3 The Algorithm

Initially, each processor contains an input integer from the range  $1 \dots \sqrt{N}$ . When the algorithm completes inputs are sorted according to row-major order. More formally, the

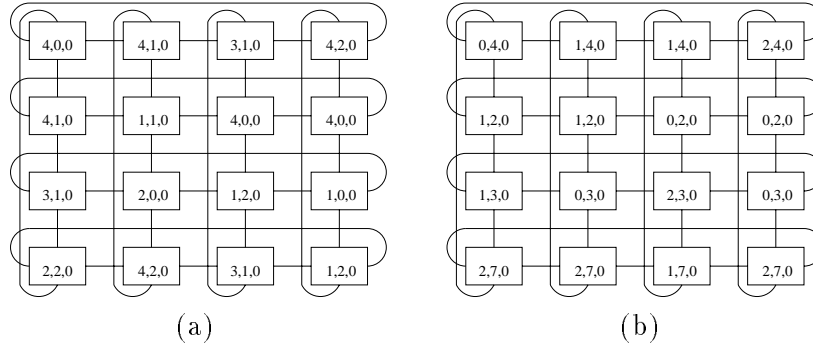


Figure 3: Configuration of registers (a) after vertical counting step, and (b) after calculating  $\text{NUMBER}(i)$ .

$(i, j)$ th processor will contain the  $(i + (j - 1) * \sqrt{N})$ th smallest element.

The idea underlying count-sort is to use the knowledge that the input range is “small” to replace the compare-exchange schemes of mesh sorts for arbitrary input with an efficient scheme to count occurrences of every possible input.

Each processor has three registers, *scratch*, *count*, and *output*. The register *scratch* holds inputs. Both *count* and *output* are initialized to 0. See Figure 2 for an example of an initial configuration on a  $4 \times 4$  mesh.

Before we proceed we need two definitions. Define  $\text{NUMBER}(i)$  to be the number of occurrences of each input value equal to  $i$ , and  $\text{LEADER}(i) = \sum_{j=1}^i \text{NUMBER}(j)$ .

For example, if we have the list 2 1 3 2 1 3 1 2 2 then

$$\text{NUMBER}(1) = 3, \text{NUMBER}(2) = 4, \text{and } \text{NUMBER}(3) = 2,$$

and

$$\text{LEADER}(1) = 3, \text{LEADER}(2) = 7, \text{and } \text{LEADER}(3) = 9.$$

Notice that if the list above is sorted to 1 1 1 2 2 2 2 3 3, then  $\text{LEADER}(i)$  is the position for the last occurrence of each  $i$ .

### 3.1 The Simple Model

We describe the five stages of count-sort in the next five subsections, and in the sixth subsection we give an analysis.

#### 3.1.1 Vertical Counting

In this first stage, processor  $(i, j)$  counts occurrences of input  $i$  in column  $j$ . To accomplish this, use the mesh connections to fully “rotate” the the input values around each column in  $\sqrt{N}$  steps (see Figure 3(a)):

```

for  $\sqrt{N}$  steps do
  if (scratch =  $x$ ) then count = count + 1
  SEND_S[scratch]

```

**Analysis:**  $\sqrt{N}$  route steps,  $\sqrt{N}$  comparisons,  $\sqrt{N}$  assignments, and  $\sqrt{N}$  increments requiring  $(\sqrt{N})t_s + (3\sqrt{N})t_o$  time steps.

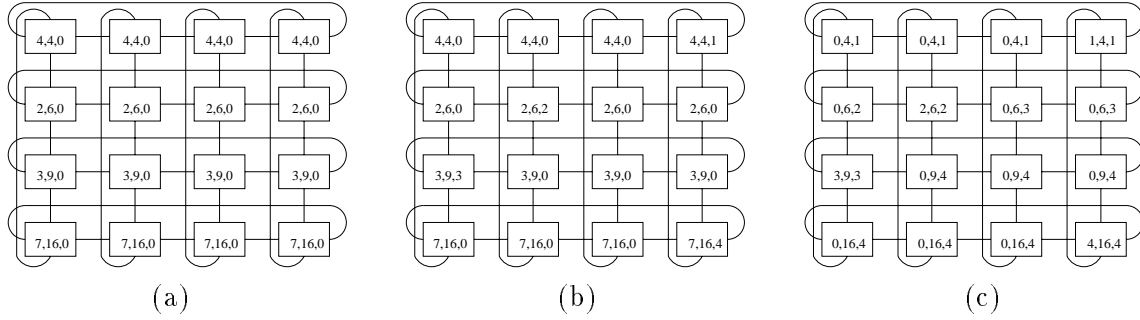


Figure 4: (a) Configuration of registers (a) after calculating  $\text{LEADER}(i)$ , (b) after routing  $i$  to processor  $\text{LEADER}(i)$ , and (c) after final stage.

### 3.1.2 Calculating $\text{NUMBER}(i)$

At this point each processor contains a partial count of the input values. Clearly, summing the contents of *count* across processors of row  $i$  will compute  $\text{NUMBER}(i)$ .

This is nearly identical to the vertical counting step, but we route horizontally and perform an unconditional addition between *scratch* and *count* (see Figure 3(b)):

```

scratch = count
for  $\sqrt{N} - 1$  steps do
    SEND_E[scratch]
    count = count + scratch

```

Now the contents of *count* in each processor of row  $i$  contains  $\text{NUMBER}(i)$ .

**Analysis:**  $\sqrt{N} - 1$  route steps,  $\sqrt{N} - 1$  increments, and  $\sqrt{N}$  assignments requiring  $(\sqrt{N} - 1)t_S + (2\sqrt{N} - 1)t_0$  time steps.

### 3.1.3 Calculating $\text{LEADER}(i)$

To calculate  $\text{LEADER}(i)$ , we perform a prefix sum down the columns. Specifically, send the contents of *scratch* vertically down the mesh performing additions between *scratch* and *count* at each step. This produces  $\text{LEADER}(i)$  in the contents of *count* across processors of row  $i$  (see Figure 4(a)):

```

scratch = count
for  $i = 1$  to  $(\sqrt{N} - 1)$  do
    SEND_S[scratch]
    if ( $y > i$ ) then count = count + scratch

```

**Analysis:**  $\sqrt{N} - 1$  route steps,  $\sqrt{N} - 1$  increments,  $\sqrt{N} - 1$  comparisons, and  $\sqrt{N}$  assignments requiring  $(\sqrt{N} - 1)t_S + (3\sqrt{N} - 2)t_0$  time steps.

### 3.1.4 Routing $i$ to Processor $\text{LEADER}(i)$

At this point, we know the value of  $\text{LEADER}(i)$ . Now, we shall send the value of  $i$  to the processor with ID  $\text{LEADER}(i)$ . To accomplish this we, again, use the mesh connections to fully “rotate” the data around the mesh. The modification in this case is that we route two values: the number  $i$  and its value  $\text{LEADER}(i)$ . The *output* registers get the value  $i$ .

Notice that it is not necessary to SEND east or west due to each column containing the same information in processors of the same row (see Figure 4(b)):

```

scratch = i
for  $\sqrt{N}$  steps do
  if (count = i + (j - 1) $\sqrt{N}$ ) then output = scratch
  SEND_S[scratch]
  SEND_S[count]

```

**Analysis:**  $2\sqrt{N}$  route steps,  $\sqrt{N}$  comparisons,  $\sqrt{N} + 1$  assignments, and  $\sqrt{N}$  additions requiring  $(2\sqrt{N})t_S + (3\sqrt{N} + 1)t_O$  time steps.

### 3.1.5 Filling in the Rest

The final step is to set *output* for processors that are between processors with ID  $\text{LEADER}(i)$ . This step completes the sorting algorithm (see Figure 4(c)):

```

for  $\sqrt{N}$  steps do
  if (j  $\neq$  1) and (output  $\neq$  0) then SEND_W[output]
  scratch = output
  if (j = 1) and (scratch  $\neq$  0) then SEND_W[scratch]
  if (j =  $\sqrt{N}$ ) and (scratch  $\neq$  0) and (i  $\neq$  1) then SEND_N[scratch]
  if (output = 0) and (scratch  $\neq$  0) then output = scratch
  if (i  $\neq$  1) then SEND_N[scratch]
  for  $\sqrt{N}$  steps do
    if (j =  $\sqrt{N}$ ) and (output = 0) then
      output = scratch
      SEND_N[scratch]
  for  $\sqrt{N}$  steps do
    if (j  $\neq$  1) and (output  $\neq$  0) then SEND_W[output]

```

**Analysis:**  $3\sqrt{N} + 3$  route steps,  $6\sqrt{N} + 8$  comparisons, and  $\sqrt{N} + 2$  assignments requiring  $(3\sqrt{N} + 3)t_S + (7\sqrt{N} + 10)t_O$  time steps.

### 3.1.6 Final Analysis

Summing the times of the five stages we get a total of time steps for count-sort:

$$(8\sqrt{N} + 1)t_S + (18\sqrt{N} + 8)t_O. \quad (1)$$

It is possible to improve the running time. Reynolds [9] points out that a slight modification to the routing stage of our algorithm in section 3.1.4 will yield a considerable speed-up as follows.

We use the fact that all processors of row *i* contain the values of  $\text{LEADER}(i)$  in *count* and *i* in *scratch*. If the processor position is less than or equal to *count* then we set *output* to *scratch*, so that we can eliminate the last stage described in Section 3.1.5. The modified fourth stage is then:

```

scratch = i
for  $\sqrt{N}$  steps do
  if (count  $\geq$  i + (j - 1) $\sqrt{N}$ ) then
    output = scratch

```

Mesh Sort	Time Steps
count-sort	$(5\sqrt{N} - 2)t_S + (11\sqrt{N} - 2)t_O$
Kumar and Hirschberg [4]	$(11\sqrt{N})t_S + (4.5 \log^2 \sqrt{N})t_O$
Nasimi and Sahni [7]	$(14(\sqrt{N} - 1) - 8 \log \sqrt{N})t_S + (6.5 \log^2 \sqrt{N} + 2.5 \log \sqrt{N})t_O$
Thompson and Kung [11]	$(14(\sqrt{N} - 1) - 8 \log \sqrt{N})t_S + (2 \log^2 \sqrt{N} + \log \sqrt{N})t_O$

Table 1: Comparing count-sort with other mesh sorts.

SEND\_S[*scratch*]  
SEND\_S[*count*]

**Analysis:**  $2\sqrt{N}$  route steps,  $\sqrt{N}$  compare steps,  $\sqrt{N} + 1$  assignment steps, and  $\sqrt{N}$  additions requiring  $(2\sqrt{N})t_S + (3\sqrt{N} + 1)t_O$  time steps.

After this modification, the algorithm is finished and the fifth stage is not needed. This improvement reduces the running time to:

$$(5\sqrt{N} - 2)t_S + (11\sqrt{N} - 2)t_O. \quad (2)$$

Compare the running time of count-sort to the running times of a few existing practical mesh sorts for arbitrary inputs that are based on the same SIMD model in Table 1.

It may be seen that, for sorting in the range  $1 \dots \sqrt{N}$ , count-sort is faster for practical  $N$ . In fact, if  $t_S$  is of the same size as  $t_O$  then count-sort is faster than the other sorts for meshes containing up to  $2^{40}$  processors, while if  $t_S \gg t_O$ , which is usually the case with real machines, count-sort is even faster.

## 3.2 Adaptation to a More Powerful model

Count-sort can be modified to run even more efficiently on our second model of computation (see Section 2.2). We examine each stage of the above algorithm to see if we are able to take advantage of the additional capabilities.

### 3.2.1 Vertical Counting

This stage remains the same.

**Analysis:**  $(\sqrt{N})t_S + (3\sqrt{N})t_O$  time steps.

### 3.2.2 Calculating NUMBER( $i$ )

We can improve this stage by observing, for this model, we need NUMBER( $i$ ) in only one column, say the first. We compute a prefix sum to the first columns in  $\frac{1}{2} \log N$  steps as follows.

We use the enhanced SEND (see Section 2.2) performing additions between processors of distances that increases by a factor of 2 (see Figure 5) until the prefix sum is computed in the first row.

$$i = 1$$

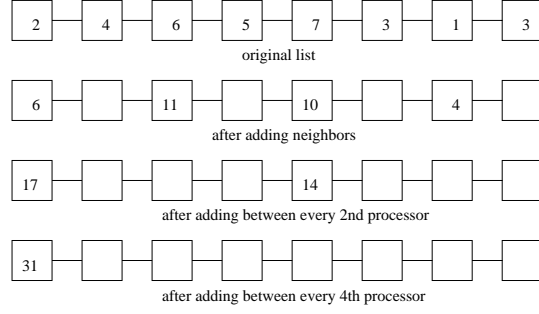


Figure 5: Logarithmic computation of  $\text{NUMBER}(i)$ .

```

scratch = counter
while  $i < \sqrt{N}$  do
  SEND[ $i$ ]W[scratch]
  counter = counter + scratch
   $i = 2 * i$ 

```

**Analysis:**  $\frac{1}{2} \log N$  enhanced SEND steps,  $\frac{1}{2} \log N$  additions,  $\frac{1}{2} \log N$  multiplications, and  $(2 + \log n)$  assignments requiring  $(\frac{1}{2} \log N)t_{SD} + (\frac{5}{2} \log N + 2)t_O$  time steps.

### 3.2.3 Calculating $\text{LEADER}(i)$

This stage remains the same.

**Analysis:**  $(\sqrt{N} - 1)t_S + (3\sqrt{N} - 2)t_O$  time steps.

### 3.2.4 Routing $i$ to Processor $\text{LEADER}(i)$

This stage is improved by routing  $i$  in a single permute step. We know the value of  $\text{LEADER}(i)$  for all  $i$ . This information is used to send each  $i$  to the position  $\text{LEADER}(i)$  with the command:

```
PERMUTE[ $i, \text{LEADER}(i)$ ].
```

**Analysis:** 1 full permutation route requiring  $t_P$  time steps.

### 3.2.5 Filling in the Rest

We can fill in the rest of the *output* registers with the SEND\_COPY command (see Section 2.2). This is performed in the same manner as the simple model, but here, instead of sending variables across the mesh with  $3\sqrt{N}$  SEND operations, we replace the latter with 3 SEND\_COPY operations.

**Analysis:** 3 SEND\_COPY steps, 14 comparisons, 3 assignments, and 3 SEND steps requiring  $3t_C + 17t_O + 3t_S$  time steps.

### 3.2.6 Final Analysis

It follows that on the improved model, the total of time steps is:

$$(2\sqrt{N} + 2)t_S + (6\sqrt{N} + \frac{5}{2} \log N + 17)t_O + (\frac{1}{2} \log N)t_{SD} + 3t_C + t_P \quad (3)$$

### 3.3 Expanding the Range

It is possible to expand the range of integers by a factor  $k$  while not increasing the running time by a factor  $k$ . To expand the range by a factor  $k$  we need  $k$  extra counter and scratch registers, following which the overall algorithm remains similar.

We achieve slowdown less than  $k$  by carefully choosing the elements to route and comparisons to make. For example, in the vertical counting stage it is possible to count  $k$  input values per processor using *no* extra route steps: simply route inputs as before, but perform  $k$  comparisons after each route step. This does not increase the number of routes, though the number of comparisons increases by a factor  $k$ . Other stages may be sped up similarly, and observe that the last two stages of the algorithm need not be altered at all for a larger range.

## 4 Implementations of Count-Sort

We implemented two versions of count-sort on a MasPar MP-1. The machine our algorithm was implemented on has 8192 processors arranged in 64 rows and 128 columns. The first implementation follows closely the modified version of the algorithm presented in Section 3.2. Even though the mesh is not square, the algorithm is essentially the same. The second implementation is a *scaled* version of count-sort. By scaled, we mean that it sorts several elements per processor.

Both implementations were written in MasPar’s Massively Parallel Language which is an extended C. The first was timed against the current library function `psort8u` for sorting 8-bit unsigned integers – at present we do not know the algorithm implemented by MasPar for `psort8u`. The second was timed against `vbsort` which is discussed in [8].

In Table 2, we give timings in number of clock ticks to sort 8-bit integers on 8192 processors. Inputs were distributed across the mesh using the pseudo-random number generator `p_random`. For each range, we ran both count-sort and `psort8u` 1000 times separately as the only job on the machine and took the average.

We were also able to obtain a very fast implementation for the scaled version of count-sort. The implementation here is for sorting large arrays of 8-bit integers. We first discuss briefly how we modified count-sort to enable it to scale to many elements per processor.

Scaling count-sort is simple. Let  $N$  be the number of elements and  $P$  be the number of processors. Note that  $N \gg P$ . We perform the counting phase in each processor for  $N/P$  steps before routing values to the next processor. This is the most time consuming step of the scaling. For large  $N/P$  it is naturally faster to route counter variables rather than input variables since the number of counters is smaller.

Calculating `NUMBER` and `LEADER` is algorithmically identical to the description given in Section 3.1. The only minor programming modification is to calculate larger values due to the large number of elements. This entails using larger variable lengths.

To route the value `LEADER` to the correct place, we call the router  $N/P$  times to send the values directly into their processor and subscript in the arrays. This involves the division of the value of `LEADER` for the correct processor and remainder arithmetic for the place in the processors’ arrays.

The final modification to count-sort is to fill in the intermediate values for the scaled model. This involves filling in values in the arrays until we reach the “bottom” of the arrays, sending values across the mesh as described originally in Section 3.1, and then filling in values from the “top” of the arrays downward until all arrays are filled.

From all this comes a very fast scaled algorithm for sorting 8-bit integers. We were able to time it against `vbsort` for 8-bit integers on a wide range of  $N/P$ , counting millions of

Range	psort8u	count-Sort	% Speed-Up
0 ... 31	31262.634	21568.108	31.0
0 ... 63	31263.178	21644.416	30.8
0 ... 127	31262.898	21727.774	30.5
0 ... 191	31263.582	21779.876	30.3
0 ... 255	31262.986	21800.624	30.3

Table 2: Comparing count-sort with the MasPar library sort psort8u counting clock ticks.

$N/P$	count-Sort	vbsort
8	3.8	2.8
256	7.2	2.7
512	7.3	2.6
1024	7.4	2.4
8192	7.5	2.2
16384	7.5	2.1
32768	7.5	2.0
61440*	7.4	1.9

\* not a power of 2 due to memory constraints

Table 3: Comparing count-sort with vbsort counting millions of instructions/second.

instructions/second. Table 3 gives a comparison of count-sort with vbsort for various sizes of  $N/P$ .

## 5 Conclusions and Future Work

We have presented a straightforward counting algorithm for sorting integers on a mesh-connected computer with  $\sqrt{N} \times \sqrt{N}$  processors, that sorts  $N$  integers in the range  $1 \dots k\sqrt{N}$  in time  $c\sqrt{N}$  where  $c$  is very small. For practical values of  $k$  and  $N$ , the algorithm proves to be very fast, both in theory and in implementation.

It is possible that this method can be expanded to sort on a larger range. One possibility is to use count-sort as a component of a parallel sorting algorithm similar to sequential radix sort. Further, the counting techniques themselves may be useful in applications other than sorting.

### Acknowledgments

We wish to thank the MasPar Corporation for allowing us to develop and test our implementation on one of their machines.

## References

- [1] T. Blank. The MasPar MP-1 Architecture. *Proc. of Compcon Spring '90*, February 1990, 20-24.

- [2] D. E. Knuth. *The Art of Computer Programming Vol. 3: Sorting and Searching*. Addison Wesley, 1973.
- [3] D. Krizanc. Integer Sorting on a Mesh-Connected Array of Processors. *Information Processing Letters*, October 1993, 283-289,
- [4] M. Kumar and D.S. Hirschberg. An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and its Application in Parallel Sorting Schemes. *IEEE Trans. on Computers*, March 1983, 254-264.
- [5] M. Kunde. Lower Bounds for Sorting on a Mesh-Connected Array of Processors. *Acta Informatica*, April 1987, 121-130.
- [6] H. Lang, M. Schimmler, H. Schmeck, and H. Schröder. Systolic Sorting on a Mesh-Connected Network. *IEEE Trans. on Computers*, July 1985, 652-658.
- [7] D. Nassimi and S. Sahni. Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Trans. on Computers*, January 1979, 2-7.
- [8] J. F. Prins and J. A. Smith. Parallel Sorting of Large Arrays on the MasPar MP-1. *Symposium on the Frontiers of Massively Parallel Computation*, October 1990, 59-64.
- [9] R. Reynolds. Personal communication.
- [10] C. Schnorr and A. Shamir. An Optimal Sorting Algorithm for Mesh Connected Computers. *Proc. 18th ACM Symp. on Theory of Computing*, 1986, 255-263.
- [11] C.D. Thompson and H.T. Kung. Sorting on a Mesh Connected Parallel Computer. *Comm. of the ACM*, April 1977, 263-271.
- [12] A. Trew and G. Wilson (Eds). *Past Present Parallel: A Survey of Available Computing Systems*. Springer-Verlag, 1991.