1995

# Communication-Efficient and Memory-Bounded External Redistribution

Jang Sun Lee
*Syracuse University*

Sanjay Ranka
*Syracuse University*, ranka@top.cis.syr.edu

Ravi V. Shankar
*Syracuse University*, rshankar@top.cis.syr.edu

# Communication-Efficient and Memory-Bounded External Redistribution

Jang Sun Lee* Sanjay Ranka Ravi V. Shankar

School of Computer and Information Science
4–116 Center for Science and Technology
Syracuse University, Syracuse, NY 13244-4100
e-mail: sunny, ranka, rshankar@top.cis.syr.edu

February 1995

**Abstract**

This paper presents communication-efficient algorithms for the external data redistribution problem. Deterministic lower bounds and upper bounds are presented for the number of I/O operations, communication time and the memory requirements of external redistribution. Our algorithms differ from most other algorithms presented for out-of-core applications in that it is optimal (within a small constant factor) not only in the number of I/O operations, but also in the time taken for communication. A coarse-grained MIMD architecture with I/O subsystems attached to each processor is assumed, but the results are expected to be applicable over a wider variety of architectures.

*This work was supported by the CASE (Computer Applications and Software Engg.) Center at Syracuse University.

# 1 Introduction

Effective parallelization of a large number of applications on coarse-grained distributed memory machines requires careful exploitation of locality to minimize communication costs. This involves distributing data structures and corresponding computations such that most computations can be performed using local data. Several distributions for arrays have been found to be useful in practice and have been incorporated into data parallel languages like High Performance Fortran [HPF94]. However, efficient data distribution for one phase of computation may in general be different from the next phase. In such cases performance improvement can be achieved by *redistribution* of data.

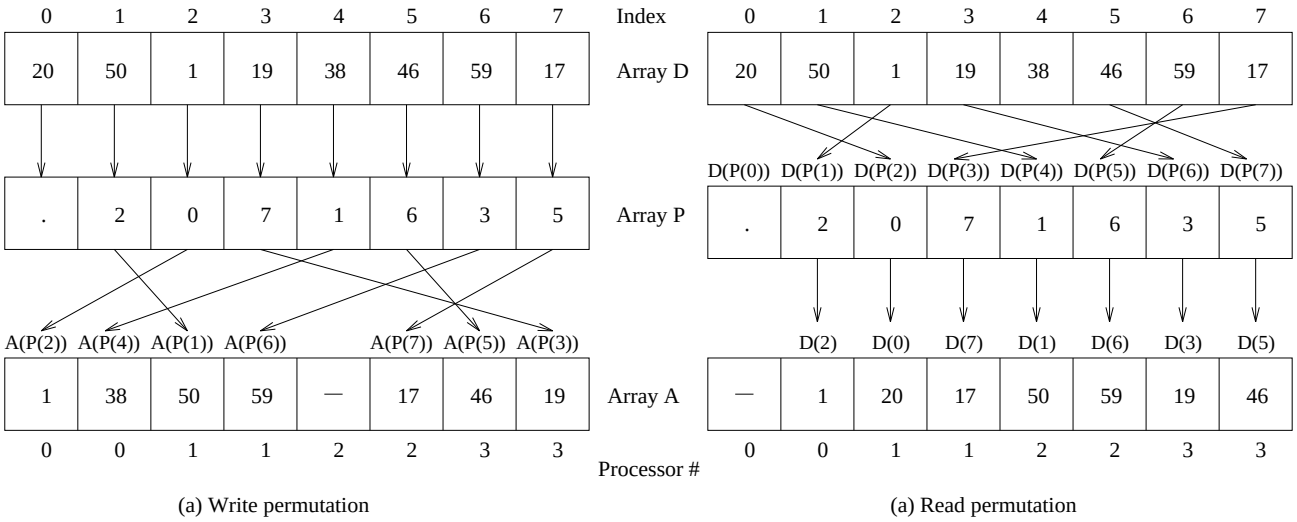
A general data *permutation* on distributed memory machines is an operation that rearranges some or all data. Redistribution of data elements from one distribution to another can be looked at as a permutation. The data movement stage in sorting algorithms, such as sample sort [ShS90, KRG94], can also be viewed as a permutation. The scalability of such applications critically depends on the cost of the permutation operation.

The throughput and capacity of Input/Output (I/O) subsystems have traditionally lagged behind corresponding increases in the speed and capacity of processors and main memory. I/O between primary and secondary memory is still a major bottleneck for most applications. To lessen the impact of this bottleneck problem, several techniques have been introduced for accessing data from the I/O subsystems of parallel machines. One such approach is to connect many disks to the processors of the parallel machine, spreading individual files across all disks to improve I/O bandwidth [Cho93, BiG88, GWR94, Kim86, PGK88, SaG86].

External (out-of-core) algorithms presented in the literature typically analyze only I/O requirements and neglect communication requirements. This is justified by the fact that current parallel systems have I/O bandwidths sufficiently lower than communication bandwidths. Vitter and Shriver [VS94] have provided an analysis of the time taken for performing a general permutation. Their algorithms are optimal in the number of I/O operations. However, the underlying parallel I/O model does not take interprocessor communication costs into account. Cormen's work [Cor93] deals with special classes of permutations such as bit-permute/complement (BPC) permutations and bit-matrix-multiply/complement (BMMC). The model used is the parallel I/O model proposed by Vitter and Shriver [VS94].

In this paper, we present external permutation algorithms that are simultaneously optimal (within a small constant factor) in terms of the number of I/O operations, communication and internal processing time. Our algorithm also has deterministic and bounded memory requirements. We expect the usefulness of our results to increase as machines with improved I/O rates become available.

The remainder of this paper is organized as follows. A general description of the permutation operation is included in the next section. Section 3 describes the architecture and storage model for the external permutation algorithms. Section 4 presents an optimal and communication-efficient




(a) Write permutation (a) Read permutation

Figure 1: Read and Write Permutations

internal permutation algorithm. A local external permutation algorithm is described in section 5. Two external permutation algorithms that incorporate the internal permutation algorithm and the local external permutation algorithm are described in section 6. This is followed by conclusions in section 7.

## 2 The Permutation Operation

Let $n$ be the number of elements distributed across $p$ processors. A permutation is an operation that rearranges data associated with some or all of the $n$ elements. Permutations can be defined as follows. Let each element $i$ $(0 \leq i < n)$ have a pointer (i.e., destination/source number) $P(i)$ $(0 \leq P(i) < n)$ and data $D(i)$ associated with it. In a *write* permutation, each element $i$ $(0 \leq i < n)$ sends its data to element $P(i)$. In a *read* permutation, each element $i$ gets data from element $P(i)$. In both cases, it is imperative that no two elements have the same value of $P(i)$.

The following code segments (in HPF) result in a write and read permutation respectively, when $P(i) \neq P(j)$ for any $i \neq j, 0 \leq i, j < n$.

```
forall (i = 0: n - 1) A(P(i)) = D(i)
forall (i = 0: n - 1) A(i) = D(P(i))
```

Figure 1 illustrates the read and write permutations. The issues involved in the design of algorithms for read permutations are very similar to those for write permutations. The rest of the paper deals with write permutations only.

## 3 Architecture and Storage Model

Communication costs as well as the number of I/O operations are being analyzed for the algorithms presented in the paper. We therefore present details about both the architecture of the coarse-grained parallel machine and the underlying disk storage model.

We model a coarse-grained parallel machine as follows. A coarse-grained machine consists of several processors connected by an interconnection network. Rather than making specific assumptions about the underlying network, we assume a two-level model of computation. The two-level model assumes a fixed cost for an off-processor access independent of the distance between the communicating processors. A unit computation local to a processor has a cost of $\delta$. Communication between processors has a start-up overhead of $\tau$, while the data transfer rate is $1/\mu$. The time taken to send a message from one processor to another is modeled as $\tau + \mu m$, where $m$ is the size of the message. For our complexity analysis we assume that $\tau$ and $\mu$ are constant, independent of the link congestion and distance between two nodes. With new techniques such as wormhole routing and randomized routing [DaS87, KRG94, Lei92, NiM93], the distance between communicating processors seems to be less of a determining factor on the amount of time needed to complete the communication. Further, the effect of link contention (due to several messages traversing common links along their routes) is limited due to the presence of virtual channels and the fact that link bandwidths are much larger than node interface bandwidths. This permits us to use the two-level model and view the underlying interconnection network as a virtual crossbar network connecting the processors. Although our algorithms are analyzed under these assumptions, they can be efficiently implemented on a wide variety of interconnection networks including meshes and hypercubes.

The overall organization of the parallel machine is shown in Figure 2. Secondary memory devices are attached directly to individual compute nodes. In this study, for simplicity, we assume that every compute node has an I/O subsystem, in contrast to more practical configurations in which only a subset of nodes host I/O subsystems. Our algorithms can be extended to deal with such scenarios. The basic storage model for external permutation shown in Figure 3. It specifies how the data elements are distributed among disks and how they are accessed by each compute node. The data assigned to each compute node is stored in a separate file called a *local file* for that node. We assume that only a portion of the local file is fetched and stored in internal memory because of the size limitation of main memory. Each compute node has sole control of the data of its disk. Any sharing of data has to be done by explicit message passing.

## 4 The Internal Permutation Algorithm

This section presents deterministic lower bounds and upper bounds on the time taken to perform an internal permutation. Both algorithms presented do not have node contention - that is, the

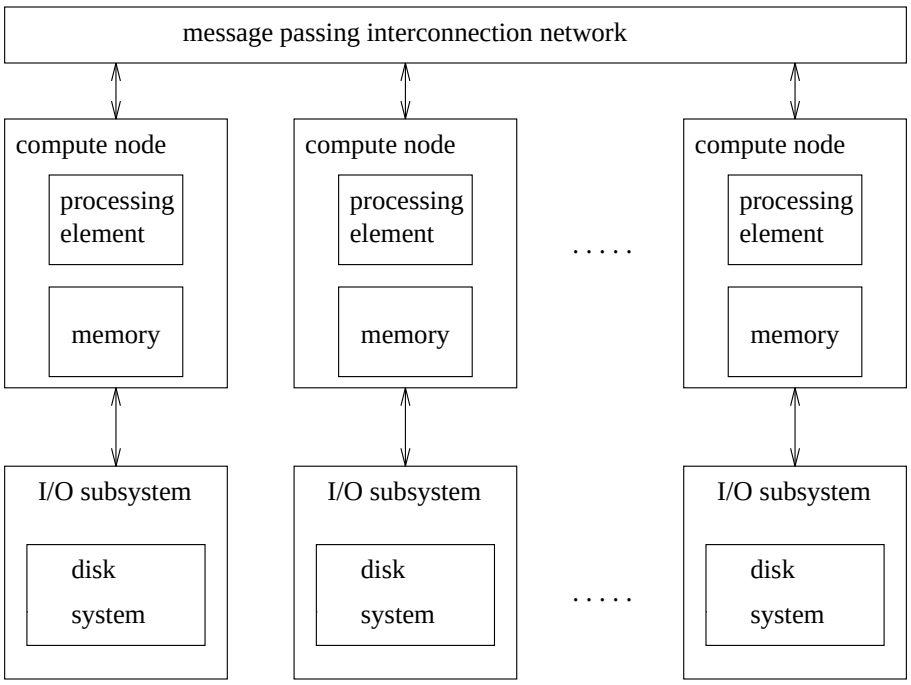



Figure 2: Architecture model

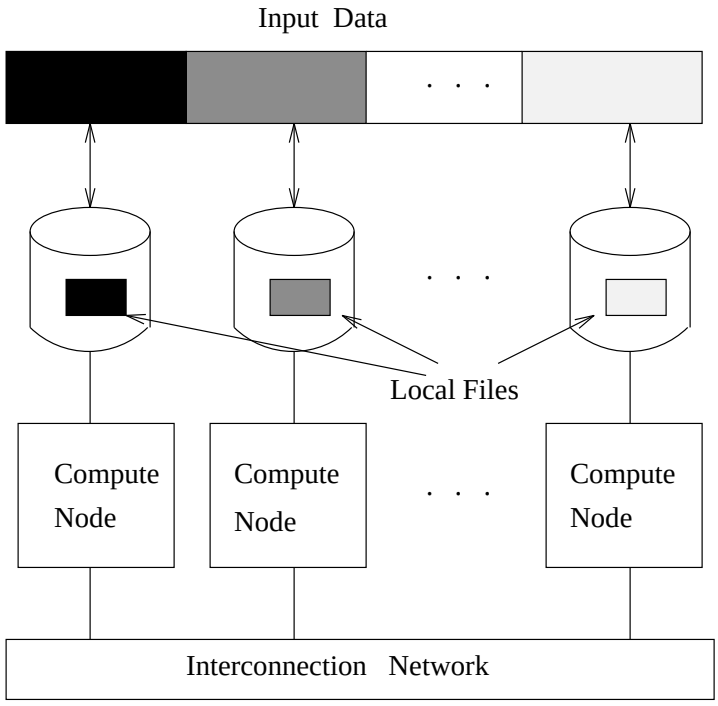



Figure 3: Storage model for external permutation

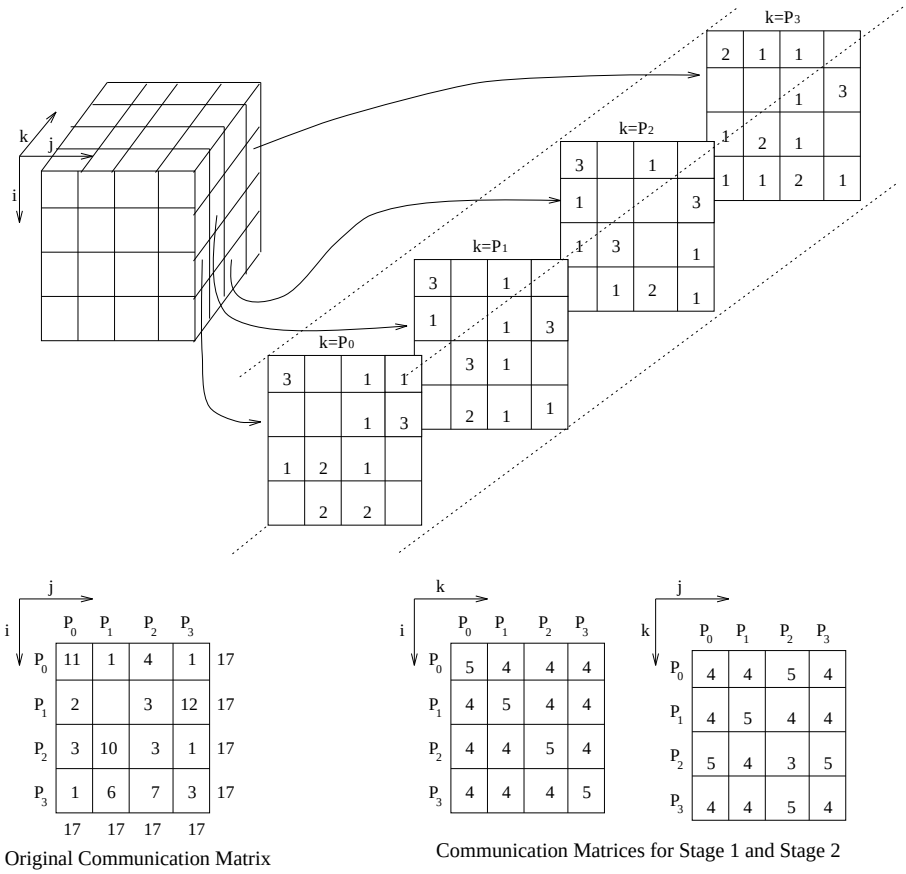



Figure 4: Splitting of messages in the permutation algorithm

algorithms ensure that no processor sends or receives more than one message at any time. During internal permutation, the sum of the sizes of messages leaving a processor, the outgoing traffic, as well as the sum of the sizes of messages entering a processor, the incoming traffic, are both upper-bounded by $t = \lceil n/p \rceil$. The permutation problem can thus be treated as a bounded *transportation* problem [SAR94]. For a machine with $p$ processors, the transportation problem can be represented using a communication matrix (a $p \times p$ matrix) $A$. The matrix entry $a_{ij}$ denotes the size of the message being sent by processor $P_i$ to processor $P_j$.

### Worst case Analysis

We have developed a two-stage algorithm that replaces the direct sending of $a_{ij}$ elements from $P_i$ to $P_j$ with sending through processors $P_k$ $(0 \leq k < p)$, which act as intermediaries [SAR94]. In stage 1, each of the $a_{ij}$ elements is divided into $p$ parts (each of size either $\lceil a_{ij}/p \rceil$ or $\lfloor a_{ij}/p \rfloor$) to be sent to processors $P_0$ to $P_{p-1}$. The splitting of messages, which is central to the algorithm, is illustrated in Figure 4. The figure also shows the original communication matrix and the communication matrices for the first and second stages, which were obtained by summation along dimensions $k$, $j$, and $i$,

respectively. The entries in the communication matrices for the first and second stages cannot be greater than $\lceil t/p \rceil$ and $\lceil t/p + p \rceil$. Local copying of data can be avoided by using send primitives that access data from noncontiguous areas (such primitives are available in standards such as MPI[MPI] and can be implemented in low-level software). Thus it can be shown that, when $t \geq O(p^2 + p\tau/\mu)$ (special cases have the traffic requirement lowered to $t \geq O(p\tau/\mu)$), communication can be performed within a factor of two of the time required for a corresponding all-to-all personalized communication with equal-sized messages. [1] The latter is a well-studied problem for which several algorithms are available in the literature using hypercubes [Bok91] (time requirements proportional to traffic) and meshes [HHK] (time requirements based on the cross-section bandwidth).

### Best case Analysis

The best case arises when the communication underlying the permutation is balanced with messages exchanged being roughly of the same size. This arises when the amount of data exchanged between any two processors does not exceed $\lceil \frac{n}{p^2} \rceil$. Only one stage of all-to-all personalized communication with equal-sized messages is needed now. For instance, a deterministic (synchronous) algorithm such as *linear permutation* can be used to perform the communication. Detection of the best case scenario is simple and requires an examination of the communication matrix. The linear permutation algorithm goes through $p - 1$ iterations, and in iteration $k$ processor $P_i$ ($0 \leq i < P$, $0 < k < P$) exchanges data with processor $P_{i \oplus k}$ ($\oplus$ is the bitwise exclusive OR operator).

The internal permutation can thus be completed in time $C\mu n/p$ ($C$ is a small constant close to 2), when $n \geq O(p^3 + p^2\tau/\mu)$. In the best case, when communication is balanced, time taken is $\mu n/p$ when $n \geq p^2\tau/\mu$. We consider the restrictions on the size $n$ of the permutation to be reasonable for the out-of-core problem being considered and do not refer to it in the rest of this paper. Details of the internal permutation algorithm can be found in [SR94a].

## 5 Local External Permutation Algorithm

In this section, we present a simple strategy for permuting data items stored on a single disk. Assume that the amount of memory in a processor is $M$ and that the size of the array to be permuted (on the single disk/compute node) is $N$.

Consider the array as being divided into $L$ runs of size $\frac{N}{L}$. The first step of the algorithm moves data items to the run they belong to. The algorithm is then applied recursively to each of the $L$ runs. We assign a bucket of size $k$ to each run in the internal memory. At each step, the algorithm reads $s$ amount of array from the disk and distributes the data to the buckets representing the $L$ runs appropriately (all elements with pointers from 1 to $\frac{N}{L}$ belong to the first run, all elements with

[1] When the communication is such that the maximum outgoing traffic at any processor is $r$ and the maximum incoming traffic at any processor is $c$, the communication takes $\mu(r + c)$ (+ lower order terms) time [SAR94].

```
// M_0 is for memoryload from the local file; M_0 = s.
// M_l is the bucket for each run l, 1 ≤ l ≤ L; M_l = k
// index i_l gives the next address in bucket l.

procedure LOCAL_EXTERNAL_PERMUTATION
begin
    i_l = 1, for each 1 ≤ l ≤ L;
    for  i = 1 to ⌈N/s⌉  do
        read s elements into M_0;   /* read from local file */
        for  j = 1 to s  do
            l = M_0[j]/(N/L);   /* find the bucket of each element */
            M_l[i_l++] = M_0[j];
            if (i_l ≥ k)
                write out M_l;    /* There are L bucket files */
                i_l = 1;
        enddo
    enddo
    for  l = 1 to L  do
        if (i_l ≥ 2)
            write out M_l;    /* remaining elements in each bucket */
    enddo
end
```

Figure 5: Local external permutation algorithm

pointers from $\frac{N}{L}+1$, ..., $2\frac{N}{L}$ belong to the second run, and so on). Whenever a bucket becomes full the data bucket is written out to the disk. The algorithm completes in $\left\lceil \frac{N}{s} \right\rceil$ steps with a computation time of $O(N)$ and I/O time equal to the reading of $\left\lceil \frac{N}{B} \right\rceil$ blocks and writing $\frac{N}{B} + L$ blocks. The detailed algorithm is shown in Figure 5.

The amount of memory required by the algorithm is $M \geq (kL + s)$. Choosing a larger value of $k$ and $s$ can potentially reduce in the total number of seeks and rotational delays if the user has control over when the data is read and written to. (We choose $s = kL$ in this paper.)

This algorithm can now be recursively applied to each of the $L$ runs. When the size of the run is less than that of the memory $M$, a local internal permutation can be performed and recursion ends. We can represent the result of the local external permutation as a complete tree with degree $L$ (see Figure 6). The nodes at level 1 of Figure 6 represent the result of the first recursion. If the size of each run at level 1 is greater than that of memory, the algorithm will be applied again to each run.

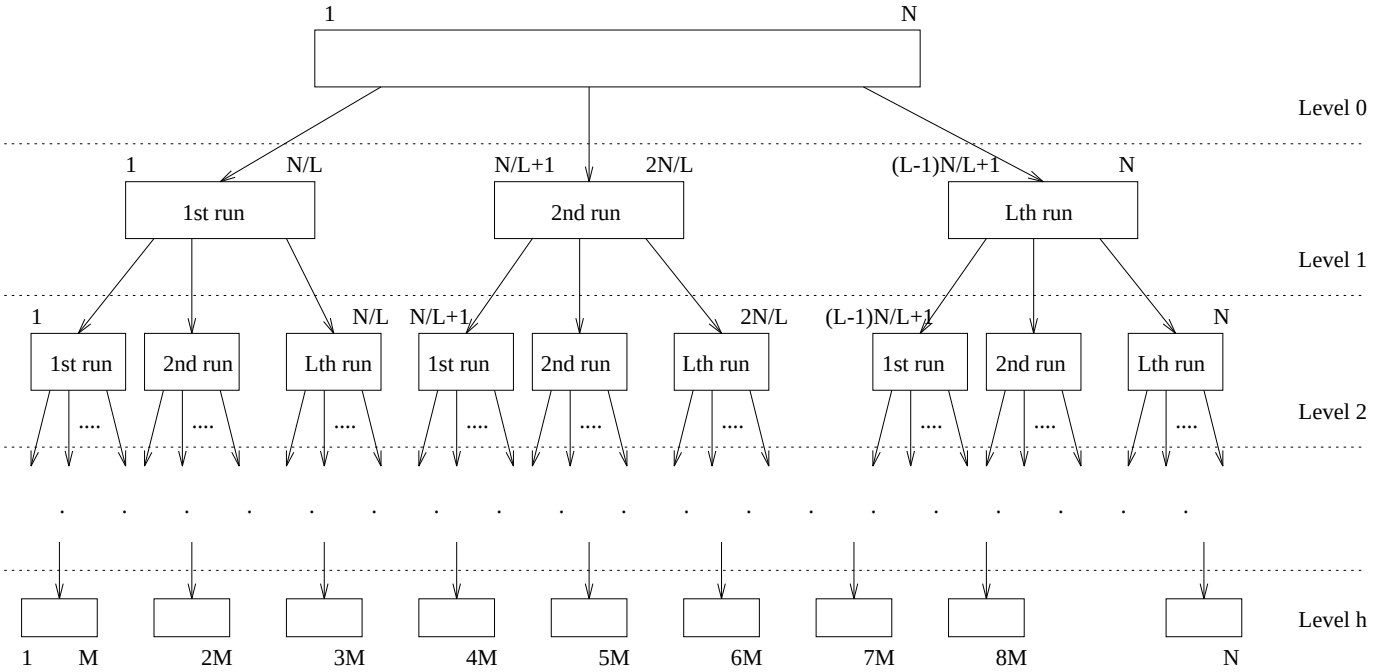



Figure 6: Local external permutation

If the size of each run at level 2 is less than that of memory, recursion ends. Assuming $M$ amount of memory, the depth of the recursion will be $\log_L \frac{N}{M}$.

An important case which is designed for the external permutation algorithm to be described later is dividing the local data items into $p$ runs. This division is based on comparison with $p-1$ keys $(n_1 \leq n_2 \leq n_3 ... \leq n_p)$. This requires $O(N)$ processing time. The number of I/O blocks read is still equal to $\left\lceil \frac{N}{B} \right\rceil$. However, the number of blocks written could be as large as $\left(\frac{N}{B} + p\right)$, while the total number of write operations could be as large as $\left(\frac{N}{k} + p\right)$.

We summarize the main performance measures of the local external permutation algorithm in Table 1.

# 6 External Permutation Algorithms

In this section, we present and analyze two algorithms for external permutation on our architecture model: *simple external permutation algorithm* and *external permutation algorithm with balanced communication*. The simple external permutation algorithm may require one less I/O pass than that the balanced external permutation algorithm. However, the incoming and outgoing traffic in the simple external permutation algorithm is not balanced and may cause bottlenecks in worst case scenarios.

We define following parameters for the architecture and storage model shown in Figure 2 and 3 for analyzing our algorithms:

Table 1: Summary of results of the local external permutation

| Terms | Dividing Data into $L$ Runs | Local External Permutation |
|---|---|---|
| Computation time | $O(N)$ | $O(N \log_L \frac{N}{M})$ |
| I/O blocks read | $\lceil \frac{N}{B} \rceil$ | $\lceil \frac{N}{B} \rceil (\log_L \frac{N}{M} + 1)$ |
| # of reads issued | $\lceil \frac{N}{s} \rceil$ | $\lceil \frac{N}{s} \rceil (\log_L \frac{N}{M} + 1)$ |
| I/O blocks written | $\frac{N}{B} + L$ | $(\frac{N}{B} + L)(\log_L \frac{N}{M} + 1)$ |
| # of writes issued | $\frac{N}{k} + L$ | $\left(\frac{N}{k} + L\right)((\log_L \frac{N}{M} + 1)$ |
| Memory constraints | $M \geq (kL + s)$ | $M \geq (kL + s)$ |

$B$: disk block size (unit of transfer between secondary and internal memory

$N$: number of data elements $M$: memory size $k$: bucket size

$s$: amount of scratchpad memory $L$: number of runs

- $P$: number of compute nodes
- $N$: number of data elements distributed across $P$ processors
- $\frac{N}{P}$: number of data elements allocated to each processor
- $M$: memory size of each processor
- $B$: disk block size (unit of transfer between secondary and internal memory)
- $s$: amount of scratchpad memory

We assume that the disk[2] attached to each processor has $\frac{N}{P}$ elements:

The disk attached to processor 1 contains elements 0, 2, ..., $\frac{N}{P} - 1$;

The disk attached to processor 2 contains elements $\frac{N}{P}, \frac{N}{P} + 1, ..., 2\frac{N}{P} - 1$;

.

.

.

The disk attached to processor $P$ contains elements $(P-1)\frac{N}{P}, (P-1)\frac{N}{P} + 1, ..., N - 1$.

### 6.1 Simple External Permutation Algorithm

Figure 7 gives a high level description of a simple external permutation algorithm on a machine with $P$ processors and $P$ disks. In each step, $s$ elements are read and distributed into $P$ buckets based

[2]We assume only one disk per processor. Our complexity analysis can be changed appropriately when the number of disks attached to a given processor is greater than 1.

```
// M_0 is for memoryload from the local file; M_0 = s.
// M_p is the bucket for processor p, 1 ≤ p ≤ P
// Memory space is allocated for the buckets
// Some parts of the memory space are allocated to each bucket when it is needed

procedure SIMPLE_EXTERNAL_PERMUTATION_ALGORITHM
begin
    for i = 1 to ⌈N/Ps⌉ do
        i_p = 1, for each 1 ≤ p ≤ P;
        read s elements into M_0;   /* local read: ⌈s/B⌉ blocks */
        for j = 1 to s do
            p = M_0[j]/(N/P);   /* find the bucket of each element */
            M_p[i_p++] = M_0[j];
        enddo
        use transportation algorithm to communicate data;
        divide data elements into L runs and store them to the disk;   /* received data elements */
        /* At processor i, the data is decomposed into L runs based
        on the first step of the local external permutation algorithm */
    enddo
    for each run (1 through L)
        apply the local external permutation algorithm
end
```

Figure 7: Simple external permutation algorithm

on the destination pointers.[3] Memory space of size $s$ is allocated for the $P$ buckets which may grow in size as needed. When the reading in of the $s$ elements and the distribution of these elements to buckets in completed, a collective communication is performed. Once the data has been received at all the processors, the data is written out to local disks in $L$ runs using the local external permutation algorithm.

### Worst case Analysis

Assuming that each step of this algorithm is executed synchronously, the outgoing traffic bound during the collective communication is $s$. In the worst case, all processors can send all data elements to the same processor. This makes $sP$ the incoming traffic bound. Thus the deterministic time

[3] In reality, this is only $P-1$ processors, since one of the buckets is being sent to source processor itself. But, in this paper, we consider the number of destination processors to be $P$, for simplicity.

requirements of this step could be as large as $(P\tau + \mu s + \mu sP)$ using the transportation algorithm [4]. Since the number of steps is $\frac{N}{Ps}$, the total communication cost can be as high as $O(\frac{N}{s}\tau + \mu N)$. The memory requirements of the above algorithm is upper bounded by $O\left(\min\left(\frac{N}{P}, Ps\right)\right)$. The I/O time on the sending site is proportional to $\lceil \frac{s}{B} \rceil$ blocks of read operations; all the reads can be performed together. However, the I/O time on the receiving processor can be as large as writing $\min\left(\frac{N}{PB}, \frac{Ps}{B}\right)$ blocks for every step. Thus, the total I/O time can be as large as writing $\min\left(\frac{N}{PB}, \frac{Ps}{B}\right) \times \frac{N}{Ps} = O\left(\min\left(\frac{N}{B}, \frac{N^2}{P^2sB}\right)\right)$ blocks. The total number of write operations to make $L$ runs at every processor can be as large as $\left(\min\left(\frac{N}{Pk}, \frac{Ps}{k}\right) + L\right)\frac{N}{Ps} = O\left(\min\left(\frac{N}{P^2ks}, \frac{N}{k}\right) + \frac{LN}{Ps}\right)$. (Memory space of size $kL$ is allocated for the $L$ buckets.)

Once the collective communication is completed, data can be locally permuted by using the *local external permutation algorithm* described in section 5. The main performance measures, under the worst case scenario, of the *simple external permutation algorithm* is summarized in Table 2. (The cost of the local external permutation is not included in this table.)

Table 2: Summary of results of the simple external permutation (worst case)

| | Sending Processor | Receiving Processor |
|---|---|---|
| Communication time | $O(\frac{N}{s}\tau + \mu N)$ | |
| Computation time | $O(\frac{N}{P})$ | $O(\frac{N}{P})$ |
| I/O blocks read | $\left\lceil \frac{N}{PB} \right\rceil$ | — |
| # of reads issued | $\left\lceil \frac{N}{s} \right\rceil$ | — |
| I/O blocks written | — | $O\left(\min\left(\frac{N}{B}, \frac{N^2}{P^2Bs}\right)\right)$ |
| # of writes issued | — | $O\left(\min\left(\frac{N}{P^2ks}, \frac{N}{k}\right) + \frac{LN}{Ps}\right)$ |
| Memory constraints | $M \geq 2s$ | $M \geq \min\left(\frac{N}{P}, Ps\right) + kL$ |

**Best case Analysis**

In the best case, the amount of data to be communicated between a pair of processors is approximately $\frac{s}{P}$ during any given phase. The time taken by the linear permutation for each of the $\frac{N}{Ps}$ steps is $P\tau + \mu s$. Thus, the total communication cost is reduced to $O(\frac{N}{s}\tau + \mu\frac{N}{P})$. The memory requirements, for the communication, would be $O(s)$ and the total I/O time would be proportional to the reading and writing of $\left\lceil \frac{N}{PB} \right\rceil$ blocks. The main performance measures, under the best case scenario, of the *simple external permutation algorithm* is summarized in Table 3. (The cost of the local external permutation is not included in this table.)

[4] No algorithm would have this requirement less than $\mu sP$ under the assumptions described in section 3.

Table 3: Summary of results of the simple external permutation (best case)

| | Sending Processor | Receiving Processor |
|---|---|---|
| Communication time | $O(\frac{N}{s}\tau + \mu\frac{N}{P})$ | |
| Computation time | $O(\frac{N}{P})$ | $O(\frac{N}{P})$ |
| I/O blocks read | $\left\lceil \frac{N}{PB} \right\rceil$ | — |
| # of reads issued | $\left\lceil \frac{N}{s} \right\rceil$ | — |
| I/O blocks written | — | $\left\lceil \frac{N}{PB} \right\rceil$ |
| # of writes issued | — | $O\left(\frac{N}{Pk}\right)$ |
| Memory constraints | $M \geq 2s$ | $M \geq 2s$ |

**Asynchronous Algorithm**

Several optimizations can be performed by running the communication and I/O asynchronously. In such an algorithm, buckets are kept for each destination in every processor. Whenever the bucket becomes full, data is sent to appropriate destination processor. At the receiving processor, the data is distributed into $L$ runs. Whenever a bucket for a run becomes full, data is written to the disk. However, there are several important factors which make it difficult to analyze the time requirements of such an external algorithm.

1. If the blocks are read such that at a given time, most of the data is destined for a particular processor, this will create hot spots and may affect the sending processors adversely because memory and communication input rate on the receiving processor is limited.

2. Asynchronous communication could cause node as well as link contention and potentially degrade overall communication time.

3. It is difficult to determine the effect of memory on the overall performance unless some assumptions are made about the data on sending site.

## 6.2 Balanced External Permutation Algorithm

In this section, we present a modification to the algorithm described in the previous subsection which reduces the memory requirements, balances communication and I/O requirements for the worst-case scenario. Further, the algorithm is close to optimal in I/O and communication requirements. The modified algorithm has three phases:

1. Rearrange the data into $P$ runs based on the keys of the data items read using the *local external permutation algorithm* described in section 5. The time requirements are proportional

```
procedure BALANCED_EXTERNAL_PERMUTATION_ALGORITHM
begin
    make P runs on the disk by applying the first step of the local external permutation algorithm
    for r = 1 to R do
        read a^r_{i1}, a_{i2}^r ..., a^r_{iP};   /* read 1/R amount of element from each of P runs */
        use transportation algorithm to communicate data;
        divide the elements into L runs and write them to the disk;    /* received data elements */
        write N/(PL) elements into L bucket files;
    enddo
    for each run (1 through L)
        apply local external permutation algorithm
end
```

Figure 8: Balanced external permutation algorithm

to $O\left(\frac{N}{P}\right)$. Let the size of run destined for processor $j$ at processor $i$ be given by $a_{ij}$.

2. The simple algorithm is applied as described in the previous subsection. However, data is read such that only a fraction $\frac{1}{R}$ of each run is read at a given time. We denote the $R$ parts of each run as $a_{ij}^r, 1 \leq r \leq R$. Figure 8 gives a high level description of this phase.

3. After completion of the colective communication, the local external permutation algorithm is applied recursively on the received data.

In phase 1 of the *balanced external permutation algorithm*, $P$ runs are created using the first stage of the local external permutation algorithm. The number of writes required to make the runs is bounded by $\left(\frac{N}{Pk} + P\right)$, assuming that a bucket of size $k$ is assigned to each run. In phase 2, $\frac{1}{R}$ amount of elements is read from each run at every iteration to communicate data. The number of read operations for the communications will be as large as $RP$. Extra I/O time equal to the reading of $\left\lceil \frac{N}{PB} \right\rceil$ blocks and writing $\frac{N}{PB} + P$ blocks is required more than the best case of the simple external permutation algorithm. This is because the simple external permutation algorithm does not start by making runs on the disk.

However, there are several important advantages of using this strategy. It is easy to show that $\sum_i \frac{a_{ij}}{R} \leq \lceil \frac{N}{PR} \rceil$ and $\sum_j \frac{a_{ij}}{R} \leq \lceil \frac{N}{PR} \rceil$, $1 \leq r \leq R$. This guarantees that the amount of data sent out or received by any processor is bounded by $\lceil \frac{N}{PR} \rceil$. The two-stage internal permutation algorithm takes time no more than $2(P\tau + t\mu)$ if the maximum outgoing and incoming traffic at any processor is $t$. Thus, the communication can be completed in $2(P\tau + \mu\frac{N}{PR})$ time in each of the $R$ iterations.

The received elements are partitioned into $L$ runs based on the keys as in the simple external permutation algorithm. Therefore, the number of I/O writes are bounded by $\frac{N}{PRk} + L$ at each of the $R$ steps. The memory requirements are upper bounded by $O\left(\frac{N}{PR}\right)$. The computation time requirements for this algorithm is $O\left(\frac{N}{P}\right)$. The main performance measures of the *balanced external permutation algorithm* is summarized in Table 4. (The cost of the local external permutation is not included in this table. The cost of the phase dividing data into $R$ runs is included.)

Table 4: Summary of results of the balanced external permutation

| | Sending Processor | Receiving Processor |
|---|---|---|
| Communication time | $O(\tau PR + \mu\frac{N}{P})$ | |
| Computation time | $O(\frac{N}{P})$ | $O(\frac{N}{P})$ |
| I/O blocks read | $2\left\lceil\frac{N}{PB}\right\rceil$ | — |
| # of reads issued | $\left\lceil\frac{N}{s}\right\rceil + RP$ | — |
| I/O blocks written | $\frac{N}{PB} + P$ | $(\frac{N}{PRB} + L) \times R$ |
| # of writes issue | $O\left(\frac{N}{Pk} + P\right)$ | $O\left(\frac{N}{Pk} + LR\right)$ |
| Memory constraint | $M \geq \max\left((s + Pk), \frac{N}{PR}\right)$ | $M \geq \frac{N}{PR} + kL$ |

# 7 Conclusions

In this paper, we have presented two communication-efficient algorithms for performing external permutations. We believe this to be one of the first attempts at performing out-of-core applications keeping all three requirements – number of I/O operations, communication time, and amount of memory needed per processor – close to optimal. Deterministic lower bounds and upper bounds were presented for the number of I/O operations, communication time and internal processing time. Communication-efficiency followed from the small constants in the time complexity of the internal permutation algorithm used.

The simple external algorithm takes communication time proportional to the total number of elements $N$ in the worst case. However, the balanced external permutation algorithm always takes time proportional to $\lceil\frac{N}{P}\rceil$, the maximum number of participating elements in each processor. It also requires less internal memory than the former algorithm.

The bounded communication and memory requirements of the balanced external permutation algorithm comes at the expense of one extra I/O phase involving the reading and writing of data. On currently available machines where the I/O bandwidth is much lower than the communication bandwidth, the simple external permutation algorithm is still likely to perform better. We expect

that in the future, machines with improved I/O rates would require the design of algorithms like the one presented to balance communication and I/O.

I/O operations can be broken into several I/O breaks in terms of disk operations. Each disk operation requires seek and rotational delay. However, if the user can perform reads and writes in arbitrary order, it can potentially reduce in the total number of seeks and rotational latency. This will also reduce the extra I/O time of the balanced external permutation algorithm. The external permutation algorithms can be extended to deal with arbitrary transportation of data. We are currently working on the design of communication-efficient algorithms for applications such as external sorting.

## References


[BiG88] D. Bitton and J. Gray, "Disk Shadowing," *Proc. Very Large Database Conf.,* 1988.

[Bok91] Shahid H. Bokhari., "Complete Exchange on the iPSC/860," ICASE Technical Report, No. 91-4, NASA Langley Research Center, Jan. 1991.

[Cho93] Alok N. Choudhary, "Parallel I/O Systems", Special Issue, *Journal of Parallel and Distributed Computing*, Vol.17, Nos.1 and 2,Jan/Feb 1993.

[Cor93] Thomas H. Cormen, "Fast Permuting on Disk Arrays", *Journal of Parallel and Distributed Computing*, Vol.17, pp.41-57, 1993.

[DaS87] William J. Dally and Chuck L. Seitz., "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers,* 36(5), May 1987, pp. 547-553.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider, "Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting," *Proc. of the Int. Conf. on Parallel and Distributed Information Systems,* Dec. 1991.

[GWR94] Gregory R. Ganger, Bruce L. Worthington, Robert Y. Hou, and Yale N. Patt, "Disk Arrays: High-Performance, High-Reliability Storage Subsystems," *IEEE Computer*, Mar. 1994, pp. 30-36.

[HHK] S.E. Hambrusch, F. Hameed, and A.A. Kohkhar, "Communication Operations on Coarse-Grained Mesh Architectures," Technical Report, Dept. of Computer Science, Purdue University.

[HPF94] High Performance Fortran Forum, *High Performance Fortran Language Specification,*, March 1994.

[Kim86] M.Y. Kim, "Synchronized Disk Interleaving," *IEEE Trans. Comput.,* Vol. C-35, Nov. 1986, pp. 978-988.

[KRG94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypsis, *Introduction to Parallel Computing: Design and Analysis of Algorithms* The Benjamin/Cummings Publishing Company, 1994.

[Lei92] C. Leiserson, et al.,"The Network Architecture of the Connection Machine CM-5," *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[MPI] MPI Forum, "The Message-Passing Interface Standard," University of Tennessee, Knoxville.

[NiM93] Lionel M. Ni and Philip K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, 26(2), Feb. 1993, pp. 62-76.

[PGK88] David Patterson, Garth Gibson, and Randy Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD Conference*, 1988, pp. 109-116.

[SaG86] Kenneth Salem and Hector Garcia-Molina, "Disk Striping," *Proc. Int'l Conf. Data Engineering*, 1986, pp. 336-342.

[SAR94] Ravi V. Shankar, Khaled A. Alsabti, and Sanjay Ranka, "The Transportation Primitive ," CIS Technical Report, Syracuse University, August 1994.
(available as ftp://top.cis.syr.edu/users/rshankar/Papers/Transportation.ps.Z)

[SR94a] Ravi V. Shankar and Sanjay Ranka, "Random Data Accesses on a Coarse-grained Parallel Machine - I. One-to-one Mappings," CIS Technical Report, Syracuse University, October 1994.
(available as ftp://top.cis.syr.edu/users/rshankar/Papers/RandomDataAccess1.ps.Z)

[ShS90] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *Journal of Parallel and Distributed Computing*, Vol. 14, 1990, pp. 361-372.

[VS94] J.S.Vitter and E.A.M Shriver, "Algorithms for parallel memory. I. Two-level memories," *Algorithmica*, vol.12, no.2-3, pp 110-47, Aug-Sept 1994.