# Parallel Merge Sort for Distributed Memory Architectures

Jean-Marc Adamo, Luis Trejo

▶ **To cite this version:**

Jean-Marc Adamo, Luis Trejo. Parallel Merge Sort for Distributed Memory Architectures. [Research Report] LIP RR-1994-05, Laboratoire de l'informatique du parallélisme. 1994, 2+35p. hal-02101948

**HAL Id: hal-02101948**

**https://hal-lara.archives-ouvertes.fr/hal-02101948v1**

Submitted on 17 Apr 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

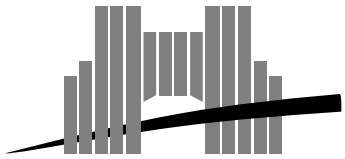

*Laboratoire de l'Informatique du Parallélisme*
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

# *Parallel Merge Sort for Distributed Memory Architectures*

Jean-Marc Adamo
Luis Trejo

January, 1994

Research Report N$^{O}$ 94-05



**Ecole Normale Supérieure de Lyon**
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

# Parallel Merge Sort for Distributed Memory Architectures

Jean-Marc Adamo

Luis Trejo

January, 1994

**Abstract**

Cole presented a parallel merge sort for the PRAM model that performs in $O(\log n)$ parallel steps using $n$ processors. He gave an algorithm for the CREW PRAM model for which the constant in the running time is small. He also gave a more complex version of the algorithm for the EREW PRAM; the constant factor in the running time is still moderate but not as small. In this paper we give an approach to implement Cole's parallel merge sort on a distributed memory architecture. Both, the CREW and the EREW algorithms have been considered. A data placement algorithm is presented as well as the associated data movements. Our proposition sorts $n$ items using exactly $n$ processors in $O(\log n)$ parallel time. The constant in the running time is only one greater than the one obtained for the PRAM model.

**Keywords:** parallel merge sort, parallel architecture, distributed memory, parallel algorithm, PRAM, pipe-line.

**Résumé**

Cole a présenté un algorithme de tri de fusion parallèle pour le modèle de calcul PRAM, qui s'exécute en $O(\log n)$ étapes parallèles en utilisant $n$ processeurs. Dans son article il donne un algorithme pour le modèle CREW PRAM, dans lequel la constante du temps d'exécution est modérée, ainsi qu'une version plus complexe pour le modèle EREW PRAM ; la constante du temps d'exécution est toujours modérée mais moins que dans la version CREW PRAM. Dans ce rapport nous donnons une approche pour l'implémentation du tri de Cole sur une architecture à mémoire distribuée. Les deux version, CREW et EREW, de l'algorithme de Cole ont été considérées. Un algorithme de placement de données est présenté ainsi que les mouvements de données associés. Notre approche permet le tri de $n$ éléments en utilisant $n$ processeurs en temps $O(\log n)$. La constante multiplicative du temps d'exécution n'est que légèrement supérieure à celle obtenue sur le modèle PRAM.

**Mots-clés:** tri de fusion, architecture parallèle, mémoire distribuée, algorithme parallèle, PRAM, pipe-line.

**Jean-Marc Adamo**

Professeur, Univérsité Claude Bernard Lyon I
I.C.S.I.
1947 Center Street. Suite 600
Berkeley, CA 94704-1105

adamo@icsi.berkeley.edu

**Luis A. Trejo R.**

Ecole Normale Supérieure de Lyon
L.I.P.
46, allée d'Italie, 69364 Lyon cedex 07, France

trejo@lip.ens-lyon.fr

# Contents

# List of Figures



# List of Tables

# 1 Introduction

Cole presented in [7] a parallel merge sort for the PRAM model that performs in $O(\log n)$ parallel steps using $n$ processors. In his paper he gave an algorithm for the CREW PRAM model for which the constant in the running time is small. He also gave a more complex version of the algorithm for the EREW PRAM; the constant factor in the running time is still moderate but not as small. In this paper we present an approach to implement both algorithms on a distributed memory architecture without incrementing significantly the constant factors in the running time. The data placement and the communications involved among processors are considered. This paper is organized as follows: Sections 2 and 4 briefly describe respectively the CREW and the EREW algorithms. Sections 3 and 5 give respectively their implementations on a distributed memory architecture.

# 2 Cole's Parallel Merge Sort: the CREW algorithm

Cole's algorithm is described in detail in [7]. It is a tree-based merge sort where the merges at different levels of the tree are pipelined. The tree is a complete binary tree where the elements to sort are initially stored at the leaves of the tree, one element per leave. For simplicity, we assume that the number of elements is a power of 2. The algorithm considers three kinds of nodes: active, inactive and complete nodes. An active node can be external or internal. At the beginning of the algorithm, all leave nodes are external with the $i$th leave containing the $i$th element of the sequence to be sorted. All other nodes are inactive. The goal of every node in the tree is to compute an ordered list $U$ comprising all elements in the subtree rooted at that node; in such a case, the node is said to be external (as for the leave nodes at the beginning of the algorithm). The algorithm of an external node is the following: if a node becomes external at stage $i$, then at stage $i+1$, it will send to its father an ordered subset of $U_i$ comprising every 4th element. At stage $i+2$ it will send to its father every 2th element of $U_i$ and at stage $i+3$ every element of $U_i$. At this moment, the node becomes a complete node and its father an external node. An inactive node becomes active (internal) when it receives for the first time a non-empty ordered list from each of its children. The algorithm of an internal node is the following: at stage $i$, the merge of the lists just received is performed to form the list $U_i$. At stage $i+1$ it will send to its father an ordered subset of $U_i$ comprising every 4th element. An ordered subset of $U$ is called a cover list of $U$ (see [7] for a formal definition of a cover list). A node remains internal until the computed $U$ list contains a sorted sequence of all elements of the tree rooted at that node. When a node becomes external at stage $i$, at stage

$i+3$ it becomes a complete node and its father an external node. Therefore, all nodes of a level $l$ become external nodes at stage $i = 3l$. As the height of the tree is $\log n$, the total number of stages performed by the algorithm is $3 \log n$. After $3 \log n$ stages the root of the tree will contain an ordered list $U$ comprising all elements of the tree.

Let us call $X_i^l$ and $Y_i^l$ the two cover lists that are received by a node of level $l$ at stage number $i$. We call $U_i^l$ the list resulting from the merge operation of the two ordered lists $X_i^l$ and $Y_i^l$. Therefore $U_i^l = X_i^l \cup Y_i^l$, where $\cup$ is the merge operation. $X_{i+1}^{l+1}$ and $Y_{i+1}^{l+1}$ are cover lists of $U_i^l$ which are sent from a node at level $l$ up to its father at level $l+1$ at stage $i+1$. Fig. 1 shows an example of the algorithm for $N = 6$, where $N$ is the height of the tree. $|Z_i|$ at level $l$ stands for the size of a $X$ (or a $Y$) list received from its left (or right) child. A node at level $l$ receives two lists ($X$ and $Y$) and performs a merge of the two to obtain the list $U$, where $|U|$ (the number of elements in $U$) is obviously the sum of the size of the lists being merged.

From Fig. 1, it is straightforward to show that a node at level $l$ becomes active (internal) at stage number $i = 2l + 1$. This comes from the fact that a node at level $l$ sends to its father for the first time a cover list of $U$ from the moment that $|U| \geq 4$. This happens after two stages the node became active (internal). As an example, consider level 4 from Fig. 1. A node at level 4 becomes active (internal) at stage $i = 9$ when it receives $|X_9| = 1$ and $|Y_9| = 1$. The merge of these two cover lists gives as a result $|U_9| = 2$. The algorithm for an internal active node is applied so an empty list is sent up since there is no 4th element in $U_9$. At stage $i = 10$, a node at level 4 receives $|X_{10}| = 2$ and $|Y_{10}| = 2$ to compute $|U_{10}| = 4$. Next, at stage 11 a cover list of $U_{10}$ ($|Z_{11}| = 1$) is sent up to level 5. Putting all pieces together, all nodes at level 4 become internal active nodes at stage $i = 9$, those of level 5 at stage $i = 11$, those of level 6 at stage $i = 13$, etc., hence the value of $i = 2l + 1$.

Even if the stage number $2l + 1$ is still valid for the two first levels, things are a little bit different. We consider the leaves of the tree as being at level 0. At the beginning of the algorithm (stage $i = 0$), all leave nodes are external nodes and $U_0$ contains the element initially stored at the leaf. Applying the algorithm of an external node, at stage $i = 1$ every 4th element is sent up to level 1 (empty list), at stage $i = 2$ every 2th (again an empty list) and it is until stage $i = 3$ that every element in the leaves is sent up to level 1. At this stage, all nodes of level 1 become active and external, since they contain all elements of the 2-leaf subtrees rooted at each node of level 1. Applying again the algorithm of an external node, at stage $i = 4$, every 4th element of $U_3$ is sent up to level 2 (an empty list), at stage $i = 5$ every 2th element of $U_3$ is sent up ($|Z_5| = 1$) so all nodes at level 2 compute $|U_5| = 2$, at stage $i = 6$ every element of $U_3$ is sent up ($|Z_6| = 2$) to compute $|U_6| = 4$ making all nodes of level 2 external. Fig. 2 illustrates a complete example for an 8-leaf binary tree.

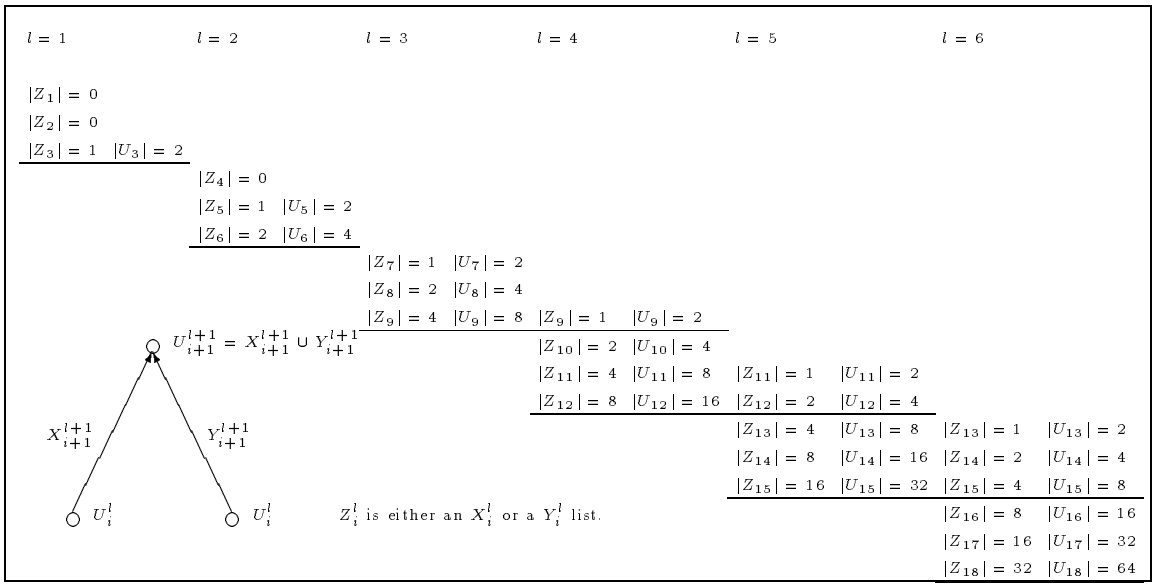



Figure 1: *Cole's Parallel Merge Sort: CREW version for $N = 6$.*

The key of Cole's algorithm relies on the merge operation of the two ordered lists, which is done in $O(1)$ parallel time. We know that the merge of two ordered lists $X$ and $Y$ of size $m$ each, if we do not have any additional information, requires $\Omega(\log m)$ parallel time with $m$ processors. Next, we will briefly describe how Cole's algorithm performs the merge operation in constant time. To do so, we need to present some definitions, taken from [7].

Consider three items $e$, $f$ and $g$ with $e < g$. $e$ and $g$ are said to *straddle* $f$ if $e \leq f < g$. Let $f$ be an element of a list $X$ and $e$ and $g$ two adjacent elements of a list $Y$ that straddle $f$. Then, the *rank* of $f$ in $Y$ is defined to be the rank of $e$ in $Y$ (in other words, the rank of an element $f$ in a list $L$ can be thought as being the number of elements in $L$ that are smaller or equal to $f$). The notation $X \rightarrow Y$ means that for each element of $X$ their ranks in $Y$ are known. Finally, $X \leftrightarrow Y$ (the cross-ranks) means that both $X \rightarrow Y$ and $Y \rightarrow X$ are known.

To perform the merge of $X_{i+1}$ and $Y_{i+1}$ at nodes of level $l$, we assume that at stage $i$ of the algorithm, the following ranks exist (see Fig. 3). By the moment, the superscript of a list variable has been left out to simplify the notation:

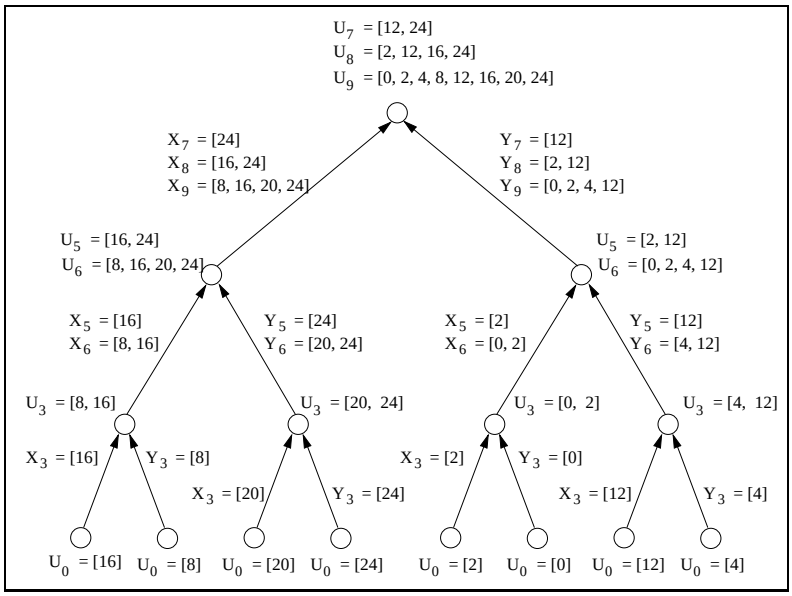



Figure 2: *Cole's Parallel Merge Sort for $N = 3$.*

$$X_i \leftrightarrow Y_i \text{ (R1)},$$
$$X_i \rightarrow X_{i+1} \text{ and } Y_i \rightarrow Y_{i+1} \text{ (R3)},$$
$$U_i \rightarrow X_{i+1} \text{ and } U_i \rightarrow Y_{i+1} \text{ (R4)}.$$

The merge proceeds in two steps: during the first one, the merge is performed in constant time by computing the cross-ranks $X_{i+1} \leftrightarrow Y_{i+1}$ (R1 in dotted lines from Fig. 3) and during the second, the ranks $U_{i+1} \rightarrow X_{i+2}$ and $U_{i+1} \rightarrow Y_{i+2}$ are maintained (R4 in dotted lines from Fig. 3) to allow the merge of the next stage of the algorithm to be performed.

*Step 1.* R1 computation. The rank of an element $e$ from $X_{i+1}$ in $U_{i+1}$ is the sum of the rank of $e$ in $X_{i+1}$ and the rank of $e$ in $Y_{i+1}$. Therefore, if we know the cross-ranks $X_{i+1} \leftrightarrow Y_{i+1}$ the merge $U_{i+1} = X_{i+1} \cup Y_{i+1}$ is performed in constant time. To compute R1, we proceed in the following way:

Consider two adjacent elements $e$ and $f$ from $U_i$. A set of elements from $X_{i+1}$ which are straddle by $e$ and $f$ is formed. The leftmost element of the set is determined by

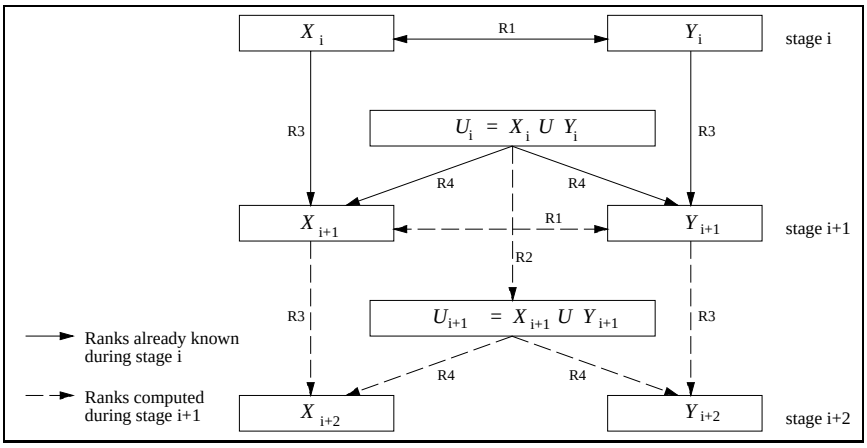



Figure 3: *Constant time merge performed at nodes of level l of the tree. Ranks computation.*

using the rank of $e$ in $X_{i+1}$ and the rightmost element by using the rank of $f$ in $X_{i+1}$. Symmetrically, a second set of elements in $Y_{i+1}$ is formed (see Fig. 4). The size of each of these sets is at most three (see [7] for properties of a 3-cover list), therefore, the cross-ranks between these two groups can be computed in constant time requiring at most 5 comparisons. The same thing is done in parallel for every pair of adjacent elements in $U_i$.

*Step 2.* R4 computation. Now, we are interested in computing the ranks $U_{i+1} \rightarrow X_{i+2}$ and $U_{i+1} \rightarrow Y_{i+2}$ which will be used to perform the merge in constant time during the next stage of the algorithm. First, we will show how the ranks R2 and R3 in dotted lines from Fig. 3 are deduced from previous ones. Since both old ranks, $U_i \rightarrow X_{i+1}$ and $U_i \rightarrow Y_{i+1}$ are known, the ranks $U_i \rightarrow U_{i+1}$ are also known (the new ranks are simply the sum of the two previous ones). Similarly, at nodes of level $l-1$ we know the ranks $U_i \rightarrow U_{i+1}$, therefore we also know the ranks $X_{i+1} \rightarrow X_{i+2}$ ($X_{i+1}$ is a cover list of $U_i$ and $X_{i+2}$ a cover list of $U_{i+1}$). In the same way, we know the ranks $Y_{i+1} \rightarrow Y_{i+2}$. Since R3 is known, for all elements in $U_{i+1}$ that came from $Y_{i+1}$ we know their ranks in $Y_{i+2}$. It remains to compute the rank in $Y_{i+2}$ of those elements in $U_{i+1}$ that came from $X_{i+1}$. We proceed in the following way (see Fig. 5):

Consider an element $e$ in $X_{i+1}$, we know the two elements $d$ and $f$ in $Y_{i+1}$ that straddle $e$ (using R1 computed during step 1). Next, if the ranks of $d$ and $f$ from $U_{i+1}$ in $Y_{i+2}$ are $r$ and $t$ respectively (using R3), we can deduce that all elements in $Y_{i+2}$ with rank less than or equal to $r$ are smaller than $e$, and those with rank greater than $t$ are greater

than $e$. Then, to compute the rank of $e$ in $Y_{i+2}$ it suffices to compute its relative order among the set of elements in $Y_{i+2}$ with rank $s$, where $r < s \leq t$. Since the maximum number of elements within this rank interval is at most 3, the relative order of $e$ can be computed in constant time using at most 2 comparisons. Symmetrically, we can compute for elements in $U_{i+1}$ came from $Y_{i+1}$ their ranks in $X_{i+2}$.

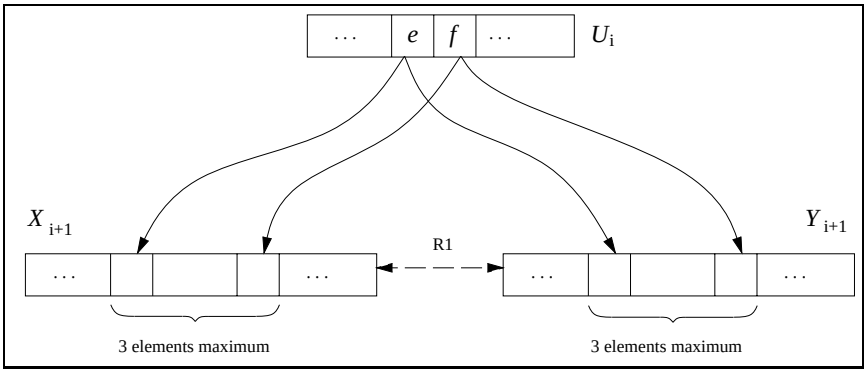



Figure 4: *Merge operation. Forming sets during R1 computation.*

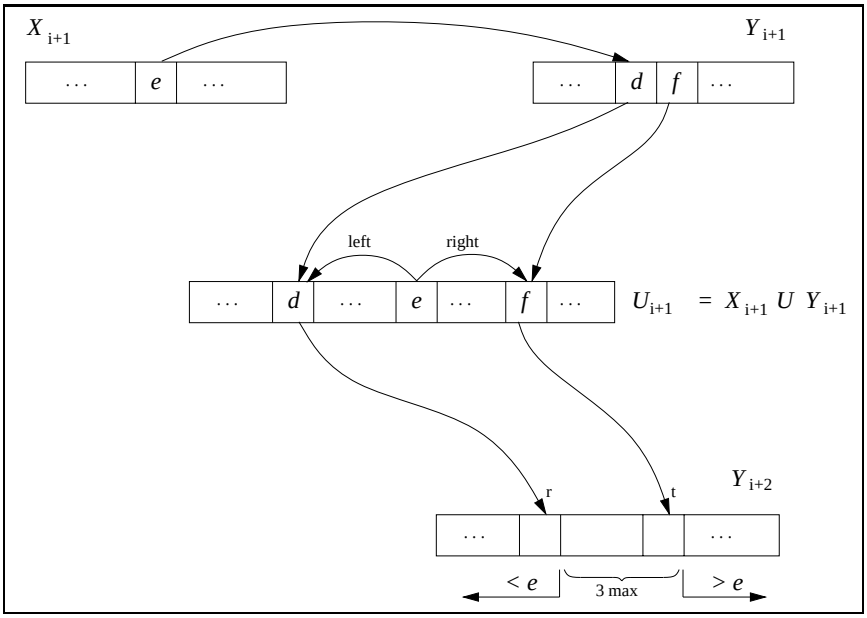



Figure 5: *Merge operation. R4 computation.*

# 3 Implementing the CREW Algorithm on a Distributed Memory Architecture

## 3.1 The Active Window

As shown in Section 2, all nodes at level $l$ of the tree become internal nodes at stage $i = 2l + 1$ and external nodes at stage $i = 3l$. We will refer to these values of $i$ as the *start_level* and the *end_level* respectively. The values of $i$ that lie in the interval [*start_level..end_level* ] represent the number of stages the nodes of a level $l$ perform a merge operation; we say that level $l$ is an *active level* during this interval. It is easy to verify that a level $l$ remains active during $l$ stages (the size of the interval); therefore, for higher levels of the tree, several levels might be active at the same stage $i$.

We will call $W_i$ the active window of the tree at stage $i$ and it is defined as the set of levels that are active at the same stage $i$ of the algorithm. As the algorithm stage number progresses, $W_i$ will change and will contain the set of levels given by the formula below.

$$W_i = \{b(i), b(i)+1, \ldots, e(i)\}$$

where

$$b(i) = \lceil \tfrac{i}{3} \rceil \text{ and } e(i) = \min\{N, \lceil \tfrac{i}{2} \rceil - 1\}.$$

The size of $W_i$ determines the number of active levels of the tree at stage $i$ and it is easily computed by the following formula. Notice that $|W_i|$ equals 0 for $i = 1$, 2 and 4.

$$|W_i| = e(i) - b(i) + 1.$$

Let us define $W_i(U)$ as the set of $U_i^l$ lists computed at every level inside $W_i$.

$$W_i(U) = \{U_i^{b(i)}, U_i^{b(i)+1}, \ldots, U_i^{e(i)}\}.$$

The size of $W_i(U)$, denoted as $|W_i(U)|$, determines the total number of elements involved in the simultaneous merge operations performed at stage $i$, and it is given by the following equation:

$$\begin{aligned} |W_i(U)| \;=\; & \{|U_i^{b(i)}| \times \text{Nn}\,(b(i)) \;+ |U_i^{b(i)+1}| \times \text{Nn}\,(b(i)+1) \;+ \ldots + \\ & |U_i^{e(i)}| \times \text{Nn}\,(e(i)) \;\} \end{aligned} \tag{1}$$

where Nn $(l) = 2^{N-l}$ is the number of nodes at level $l$. In order to compute $|U_i^l|$ we consider the following facts: at stage number $i = start\_level$, $|U_i^l| = 2$; at stage number $i = end\_level$, $|U_i^l| = 2^l$ (see Section 2). Therefore $|U_i^l| = 2^{i-start\_level+1}$. Replacing $|U_i^l| by 2^{i-2l}$ in (1), we obtain:

$$|W_i(U)| = \{2^{i-2b(i)} \times 2^{N-b(i)} + 2^{i-2(b(i)+1)} \times 2^{N-(b(i)+1)} + \ldots + 2^{i-2e(i)} \times 2^{N-e(i)}\}.$$

Let $S_i$ be equal to $|W_i(U)|$, then

$$\begin{aligned} S_i &= \sum_{k=b(i)}^{e(i)} 2^{i-2k} \times 2^{N-k} \\ &= \frac{8n}{7} \times 2^i \times \left\{ \frac{1}{8^{b(i)}} - \frac{1}{8^{e(i)+1}} \right\}. \end{aligned} \tag{2}$$

## 3.2 Data Placement

### 3.2.1 Initial Data Placement and Resource Constraints

The algorithm sorts $n$ elements using $n$ processors, initially one element per processor. In general, we assign one processor to each element of every list inside $W_i(U)$. Therefore, the number of processors needed at stage $i$ is given by $S_i$.

**Lemme 3.1** *The number of processors needed by the algorithm at stage $i$ is greater than or equal to $n$ for $i \bmod 3 = 0$ and less than $n$ for other values of $i$.*

*Proof.* Let us rewrite (2) as:

$$S_i = \frac{n}{7} \times A - \frac{n}{7} \times B \tag{3}$$

where

$$A = \frac{2^{i+3}}{2^{3b(i)}} \text{ and } B = \frac{2^{i+3}}{2^{3\times(e(i)+1)}}.$$

To compute the value of A, three cases are considered:

1. $i \bmod 3 = 0$.
   $b(i) = \frac{i}{3}$, therefore $A = 8$,

2. $i \bmod 3 = 1$.
   $b(i) = \frac{i+2}{3}$, therefore $A = 2$,
3. $i \bmod 3 = 2$.
   $b(i) = \frac{i+1}{3}$, therefore $A = 4$.

Recall that level $N$ is an active level whenever $2N + 1 \leq i \leq 3N$; therefore, for values of $i$ inside this interval, $e(i) = N$. For other values of $i$, $e(i) = \lceil \frac{i}{2} \rceil - 1$. So to compute $B$, we consider two cases:

1. for $2N + 1 \leq i \leq 3N$
   $e(i) = N$, therefore $B = \frac{2^i}{2^{3N}}$,
2. for $i < 2N + 1$, two sub-cases are considered:
   (a) $i \bmod 2 = 0$.
   $e(i) = \frac{i}{2} - 1$, therefore $B = \frac{2^3}{2^{\frac{i}{2}}}$,
   (b) $i \bmod 2 = 1$.
   $e(i) = \frac{i+1}{2} - 1$, therefore $B = \frac{2^{\frac{3}{2}}}{2^{\frac{i}{2}}}$.

It is clear that B is always greater than 0. Hence $S_i < n + \frac{n}{7}$ for $i \bmod 3 = 0$, $S_i < \frac{2}{7} \times n$ for $i \bmod 3 = 1$ and $S_i < \frac{4}{7} \times n$ for $i \bmod 3 = 2$. We still have to prove that $S_i \geq n$ for $i \bmod 3 = 0$. Replacing $A$ in (3) we obtain:

$$S_i = n + \frac{n}{7} - \frac{n}{7} \times B. \tag{4}$$

In order to determine the lower bound of $S_i$ we have to know the values of $i$ ($i \bmod 3 = 0$) for which $B$ is maximum. From $B$ computation first case, $B$ is maximum when $i$ takes the greatest possible value. This value is $3N$. From the second case, $B$ is maximum when $i$ has the smallest possible value. This is obviously equal to 3, since it is the smallest value of $i$ that satisfies the initial condition $i \bmod 3 = 0$. Thus, we can conclude that the maximum possible value of $B$ is 1. Replacing $B$ in (4) we obtain $S_i = n$, therefore, $n \leq S_i < n + \frac{n}{7}$. □

So far, we have shown that whenever $i \bmod 3 = 0$ more processors than the available will be needed to compute $W_i(U)$. For other values of $i$, the number of processors required will be smaller than the processors at hand. We will show in the next section that this resource constraint will imply an extra parallel step to be performed without incrementing significantly the time complexity of the algorithm.

### 3.2.2 $W_i(U)$ Data Placement

The data placement for $W_i(U)$ is known if the data placement of each of the $U_i^l$ lists inside $W_i(U)$ is known. So next, we will show how data of any $U_i^l$ list is distributed among the $n$ processors. The set of processors is originally arranged as a logical linear array and each processor is numbered sequentially from 0 to $n-1$.

#### $U_i^l$ Data Placement

The $X$ and $Y$ lists are composed of the following arguments:

$X(i, l, p, s, d, \mathit{offset})$, $Y(i, l, p, s, d', \mathit{offset})$, where

- $i$ stands for the main algorithm stage number,
- $l$ stands for the level in the tree,
- $p$ stands for the size of the list (number of processors),
- $s$ stands for the data placement step,
- $d$ is the first processor from which the step $s$ is applied,
- *offset* is used when spreading the data.

The exact definition and computation of $s$, $d$ and *offset* will be given in due course. The notation used so far $(X_i^l, Y_i^l)$ is in fact a short form of the previous one. $Z_i^l$ will refer either to a $X_i^l$ list or to a $Y_i^l$ list. At stage $i$ of the algorithm, a node at level $l$ receives $X_i^l$ and $Y_i^l$ so the computation of $U_i^l$ is performed:

$$X(i, l, p, s, d, \mathit{offset}) \cup Y(i, l, p, s, d', \mathit{offset}) = U(i, l, 2p, s, d, \mathit{offset})$$

$$\text{for } \mathit{start\_level} \leq i \leq \mathit{end\_level}.$$

The size of $U_i^l$ is the sum of the size of the two lists involved in the merge operation. Its $s$, $d$ and *offset* arguments are equal to $s$, $d$ and *offset* in $X_i^l$. At stage $i+1$, a cover list, either a $X_{i+1}^{l+1}$ or a $Y_{i+1}^{l+1}$ list, is formed from $U_i^l$ and sent to level $l+1$:

$$U(i, l, 2p, s, d, \mathit{offset}) \overset{\text{cover list}}{\longrightarrow} Z(i+1, l+1, \tfrac{2p}{r}, s, d, \mathit{offset})$$

$$\text{for } start\_level + 1 \ \leq i \leq end\_level + 3.$$

The size of $Z$ is given by its third argument, which recalls that $Z$ is a cover list comprising every $r$th element of $U$: $r$ equals 4 for $start\_level + 1 \leq i \leq end\_level + 1$ ($Z$ is empty for $i = start\_level + 1$); $r$ equals 2 for $i = end\_level + 2$ and $r$ equals 1 for $i = end\_level + 3$. The arguments $s, d$ and *offset* are computed from the new values of $i$ and $l$. The computation of the arguments $i$, $l$ and $p$ have already been presented. By the moment, we will show how to compute $s$ and $d$. The use of *offset* and its computation will be explained later.

To compute $s$ and $d$, we recall the fact that at each of the $2^{N-l}$ nodes of an active level $l$, there is a $U_i^l$ list. We have seen that the number of elements (therefore the number of processors needed) at level $l$ is given by the product $|U_i^l| \times 2^{N-l}$. Since the data is distributed uniformly among the $n$ processors, dividing $n$ by the last product we obtain the data placement step:

$$s = \frac{\text{Total number of processors}}{\text{Size of } U_i^l \times \text{ Number of } U_i^l \text{ lists}}$$

$$s = \frac{n}{2^{i-2l} \times 2^{N-l}} = 2^{3l-i}.$$

Now, using the data placement step and the size of a $U_i^l$ list, it is easy to determine the value of the $d$ argument, that is, the processor number which holds the first element of the list: at level $l$, the first $U_i^l$ list will start at processor $d = 0$, the second $U_i^l$ list at processor $d = s \times 2p$, the third one at processor $d = s \times 2p + s \times 2p$, etc. In a similar way, we can see that the first $X_i^l$ list will start at processor $d = 0$, the first $Y_i^l$ list will start at processor $d' = s \times p$, the second $X_i^l$ list will start at processor $d = s \times p + s \times p$, the second $Y_i^l$ list will start at processor $d' = s \times p + s \times p + s \times p$, etc. (see Fig. 6). Hence, the arguments $d$ and $d'$ are computed using the formulas below.

$$\begin{aligned} d &= s \times p \times 2k && \text{for } 0 \leq k < 2^{N-l} \\ d' &= s \times p \times (2k+1) && \text{for } 0 \leq k < 2^{N-l}. \end{aligned}$$

What we are looking for is to perform the computation of $W_i(U)$ in one parallel step using $n$ processors. It is clear that the data placement used so far for each of the $U_i^l$ lists inside $W_i(U)$ results in sharing some processors. To avoid these processor conflicts and to achieve maximum parallelism, the lists have to be spread among the available processors. Next, we will present the data placement algorithm which satisfies the last conditions. The algorithm relies on the *offset* argument computation.

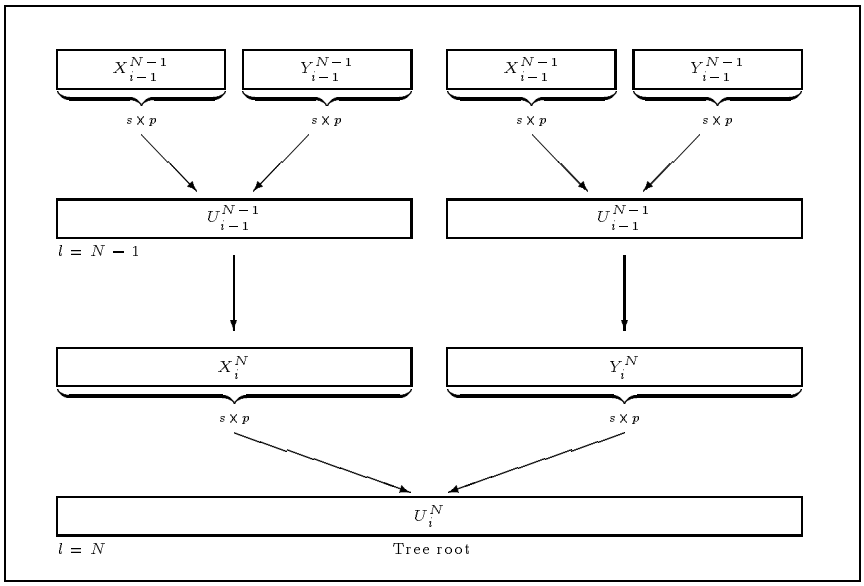



Figure 6: *d Computation.*

**Data Placement Algorithm**

The data placement algorithm to spread the lists inside $W_i(U)$ among the available processors is very simple: the $U_i^l$ lists of the same level are shifted to the right a number of processors given by the value of its *offset* argument. This shift is obtained by incrementing $d$ ($d = d + offset$). Below, we show the values of the argument *offset* for each of the lists inside $W_i(U)$. Two cases are considered:

1. $(i \bmod 3) \neq 0$ then

   $U_i^{b(i)}$ $\quad offset = 0,$

   $U_i^{b(i)+1}$ $\quad offset = 1,$

   $U_i^{b(i)+2}$ $\quad offset = 3,$

   $U_i^{b(i)+3}$ $\quad offset = 5, etc.$

2. $(i \bmod 3) = 0$

   $U_i^{b(i)}$ $\quad offset = 0,$

Since $U_i^{b(i)}$ is computed using exactly $n$ processors, the next $U_i^{b(i)+1}, U_i^{b(i)+2}, \ldots, U_i^{e(i)}$ computations are performed in an extra parallel step using less than $\frac{n}{7}$ processors (see Lemma 3.1). Hence the algorithm resumes in the following way:

$U_i^{b(i)+1}$ $\quad offset = 0,$

$U_i^{b(i)+2}$ $\quad offset = 1,$

$U_i^{b(i)+3}$ $\quad offset = 3,$

$U_i^{b(i)+4}$ $\quad offset = 5, etc.$

The *offset* argument of $U_i^l$ is computed by the following function (see Table I):

---

```
offset (U_i^l)
    /* pos is the rank of U_i^l in W_i(U) */
    pos = l − b(i)
    if (i mod 3) ≠ 0) then
        if pos = 0 then
            offset = 0
        else
            offset = 2 × pos − 1.
        endif
    else
        if pos = 0 or pos = 1 then
            offset = 0
        else
            offset = 2 × pos − 3.
        endif
    endif
end
```

---

Next we will prove that shifting the lists in this way will not create any conflict, like having more than one element assigned to the same processor at the same time, for a given problem size. Let $W_i^0(U)$ be $W_i(U)$ for $i \bmod 3 = 0$, similarly $W_i^1(U)$ for $i \bmod 3 = 1$ and $W_i^2(U)$ for $i \bmod 3 = 2$. The computation of $W_i^0(U)$ is performed in two parallel steps: the first one using $n$ processors to compute $U_i^{b(i)}$ and the second one to compute $U_i^{b(i)+1}, U_i^{b(i)+2}, \ldots, U_i^{e(i)}$ using less than $\frac{1}{7} \times n$ processors. $W_i^1(U)$ is computed in one parallel step using less than $\frac{2}{7} \times n$ processors and $W_i^2(U)$ in one parallel

| | $U_i^{b(i)}$ | $U_i^{b(i)+1}$ | $U_i^{b(i)+2}$ | $U_i^{b(i)+3}$ | $U_i^{b(i)+4}$ | ... | $U_i^{e(i)}$ |
|---|---|---|---|---|---|---|---|
| *pos:* | 0 | 1 | 2 | 3 | 4 | ... | $e(i) - b(i)$ |
| $(i \bmod 3) \neq 0$ *offset:* | 0 | 1 | 3 | 5 | 7 | ... | $2 \times pos - 1$ |
| $(i \bmod 3) = 0$ *offset:* | 0 | 0 | 1 | 3 | 5 | ... | $2 \times pos - 3$ |

Table I: *Computation of the offset argument of a $U_i^l$ list.*

step using less than $\frac{4}{7} \times n$ processors. Shifting the lists is only done during the second step of $W_i^0(U)$ computation, and during $W_i^1(U)$ and $W_i^2(U)$ computation. The set of processors assigned by the algorithm to compute the second step of $W_i^0(U)$ and the one to compute $W_i^1(U)$ are each a subset of the set of processors assigned to compute $W_i^2(U)$. Therefore, in order to prove that the data placement algorithm creates no conflict, the analysis of $W_i^2(U)$ computation suffices.

Fig. 7 depicts the data placement algorithm. It is easy to verify that, from $W_i^2(U)$, the data placement step $s$ for $U_i^{b(i)}, U_i^{b(i)+1}, U_i^{b(i)+2}$, etc. is $2^1, 2^4, 2^7$, etc. respectively. Their *offset* values are 0, 1, 3, etc. respectively. Hence, $U_i^{b(i)}$ will be placed every $2^1$ processors starting at processor $d$ + *offset*: $\{0, 0+2^1, 0+2^1+2^1$, etc.$\}$ making all even processors taken. $U_i^{b(i)+1}$ will be placed every $2^4$ processors starting at processor $d$ + *offset*: $\{1, 1+2^4, 1+2^4+2^4$, etc.$\}$ allocating only odd processors. $U_i^{b(i)+2}$ will be placed every $2^7$ processors starting at processor $d$ + *offset*: $\{3, 3+2^7, 3+2^7+2^7$, etc.$\}$ allocating also odd processors. $P^k$ stands for the set of processor numbers holding $U_i^{b(i)+k}$.

**Lemme 3.2** *The data placement algorithm generates a set of processor sequences with no conflict (i.e. no processors in common between any two sequences) for a problem size of* $\log n < 29$.

*Proof.* The processor sequences generated by the algorithm for $W_i^2(U)$ can be expressed by the equations below.

$$P^0 = 0 + 2^1\alpha$$

| | | | *Processor Number* | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $W_i^2(U)$ | step $s$ | $P^k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| $U_i^{b(i)}$ | $2^1$ | $P^0$ | x | | x | | x | | x | | x | | x | | x | | x | | x | | x |
| $U_i^{b(i)+1}$ | $2^4$ | $P^1$ | | x | | | | | | | | | | | | | | | | x | |
| $U_i^{b(i)+2}$ | $2^7$ | $P^2$ | | | | x | | | | | | | | | | | | | | | |
| $U_i^{b(i)+3}$ | $2^{10}$ | $P^3$ | | | | | | x | | | | | | | | | | | | | |
| $U_i^{b(i)+4}$ | $2^{13}$ | $P^4$ | | | | | | | | x | | | | | | | | | | | |
| $U_i^{b(i)+5}$ | $2^{16}$ | $P^5$ | | | | | | | | | | x | | | | | | | | | |
| $U_i^{b(i)+6}$ | $2^{19}$ | $P^6$ | | | | | | | | | | | | x | | | | | | | |
| $U_i^{b(i)+7}$ | $2^{22}$ | $P^7$ | | | | | | | | | | | | | | x | | | | | |
| $U_i^{b(i)+8}$ | $2^{25}$ | $P^8$ | | | | | | | | | | | | | | | | x | | | |
| $U_i^{b(i)+9}$ | $2^{28}$ | $P^9$ | | | | | | | | | | | | | | | | | | x | |

Figure 7: *Data placement algorithm.*

$$P^1 = 1 + 2^4\alpha$$

$$P^2 = 3 + 2^7\alpha$$

$$P^3 = 5 + 2^{10}\alpha$$

$$P^4 = 7 + 2^{13}\alpha$$

$$\vdots$$

$$P^8 = 15 + 2^{25}\alpha$$

for $\alpha \in I\!N$.

As remarked previously, $P^0$ allocates all even processors, and $P^k$, for $k = 1..8$ allocates only odd processors. Hence $P^0$ and $P^k$ are disjoint. Next, we still have to prove that

$P^k$ and $P^{k'}$ are disjoint for any $k$ and $k'$ having $k \neq k'$. $P^k$ can be expressed by the following equation:

$$P^k = (2k-1) + 2^{3k+1}\alpha \text{ for } \alpha \in I\!N \text{ and } k = 1..8.$$

For any $k$ and $k'$ so that $k \neq k'$, we will prove that there is no conflict between $P^k$ and $P^{k'}$. To do so, we only need to prove that the next equation:

$$(2k-1) + 2^{3k+1}\alpha = (2k'-1) + 2^{3k'+1}\alpha' \tag{5}$$

leads to a contradiction.

Eq. (5) can be written as follows:

$$\alpha' = \frac{k-k'}{8^{k'}} + \alpha 8^{k-k'}$$

If $k > k'$

$k - k' \leq 7$ and $8^{k'} \geq 8$, implies that $\frac{k-k'}{8^{k'}} \notin I\!N$, therefore $\alpha' \notin I\!N$. Then Eq. (5) leads to a contradiction.

If $k < k'$

$$\alpha' = \frac{\alpha}{8^{k'-k}} - \frac{k'-k}{8^{k'}}$$

where $k' - k = 1, \ldots, 7$ and $k' = 2, \ldots, 8$. If $\alpha' \in I\!N$, there exist two numbers $\alpha, q \in I\!N$ such that:

$$\alpha = 8^{k'-k}q + \frac{k'-k}{8^{k'}}$$

that is to say, such that:

$$8^{k'}(\alpha - 8^{k'-k}q) = k' - k$$

therefore $\alpha - 8^{k'-k}q$ should be fractional, which is not possible.

Finally, as remarked in Fig. 7, applying $P^9$ would generate the first processor conflict: the first processor allocated by $P^9$ will conflict with the second processor assigned by $P^1$.

$$P^9 = 17 + 2^{25} \times 0 = 17$$
$$P^1 = 0 + 2^4 \times 1 = 17.$$

$P^9$ is applied when $e(i) - b(i) \geq 9$. The value of $i$ that satisfies the previous relation is $i \geq 59$. Replacing $i = 59$ in $U_i^{e(i)}$ we obtain a level $l = 29$, representing a problem size of at least $n = 2^{29}$ elements (536,870,912 processors), which is quite impractical by the moment. □

Let us define *argument (List)* as a function returning the argument *argument* of the list *List*. As already mentioned, $U_i^{b(i)+k}$ will be assigned a set of processors given by $P^k$. The following 2 properties can easily be verified:

1. A subset of $P^k$ is obtained by shifting $P^{k'}$ *offset* $(U_i^{b(i)+k'})$ – *offset* $(U_i^{b(i)+k})$ processors to the left, where $k < k'$.
2. A superset of $P^{k'}$ is obtained by shifting $P^k$ *offset* $(U_i^{b(i)+k'})$ – *offset* $(U_i^{b(i)+k})$ processors to the right, where $k < k'$.

As it has surely been noticed, shifting the lists implies a data arrangement to be performed. The data movements required are explained in the following section.

## 3.3 Data Movements

The function *Merge_with_help* $(U_i^l, X_i^l, Y_i^l, U_{i-1}^l)$ presented in Section 3.4, merges the lists $X_i^l$, $Y_i^l$ with help of $U_{i-1}^l$ in order to compute $U_i^l$ (see Section 2). So in order for a merge to take place, the following lists have to reside on the same set of processors: $U_{i-1}^{l-1}$, from which the cover list $Z_i^l$ is formed, and $U_{i-1}^l$.

Two kinds of data movements are needed to guarantee the correct placement of the lists; these movements are presented below.

## Positive shift

The first data movement required comes from the need of having all elements of $U_i^l$ reside on the same set of processors as the one holding $Z_{i+1}^{l+1}$. As already mentioned, $Z_{i+1}^{l+1}$ is a cover list of $U_i^l$. Then, in order to have $U_i^l$ reside on the right place, all elements of $U_i^l$ have to be shifted $offset\ (Z_{i+1}^{l+1}) - offset\ (U_i^l)$ processors to the right. We will call this data movement a **positive shift**. It is straightforward to verify that there is a positive shift equal to 0 when $U_i^l$ belongs to $W_i^2(U)$ and whenever $i = 3l$ ($U_i^{b(i)}$ from $W_i^0(U)$); in such cases, there is not data movement. For all other cases, the positive shift has a value greater than 0, in which case the data movement is performed only if the size of $U_i^l$ is greater than or equal to 4. The following communication function performs such a positive shift:

---

```
+Shift (U_i^l)
    if (l + 1 ≤ N) then
        Comm_offset = offset (Z_{i+1}^{l+1}) - offset (U_i^l)
        if (Comm_offset > 0) and (p (U_i^l) ≥ 4) then
            Send (+, U_i^l, Comm_offset)
        endif
    endif
end
```

---

The function *Send*, sends every element of $U_i^l$ to processor $P$+*Comm_offset*, where $P$ is a processor holding an element of $U_i^l$. The first argument indicates that the send operation corresponds to a positive shift.

## Negative shift

The second data movement operation comes from the need of having $U_i^l$ reside on the same set of processors holding $X_{i+1}^l$ and $Y_{i+1}^l$, so the merge *Merge_with_help* ($U_{i+1}^l$, $X_{i+1}^l$, $Y_{i+1}^l$, $U_i^l$) can take place. In order to have $U_i^l$ reside on the right place, all elements of $U_i^l$ have to be shifted $offset\ (U_i^l) - offset\ (Z_{i+i}^l)$ processors to the left. We will call this data movement a **negative shift**. It is easy to verify that there is a negative shift to be performed whenever $U_i^l$ belongs to $W_i^2(U)$, except for $i = 3l-1$ ($U_i^{b(i)}$ from $W_i^2(U)$). The value of the negative shift in such a case and in all other cases is equal to 0, so no data movement is performed. The following communication function performs such a negative shift:

---

$-Shift\ (U_i^l)$
    $Comm\_offset = offset\ (U_i^l)$ - $offset\ (Z_{i+1}^l)$.
    if ($Comm\_offset > 0$) then
        $Send$ (-, $U_i^l$, $Comm\_offset$)
    endif
end

---

The first argument of the *Send* function indicates that the send operation corresponds to a negative shift, so every element of $U_i^l$ is sent to processor $P-Comm\_offset$, where $P$ is a processor holding an element of $U_i^l$. As a final remark, it is important to notice that whenever a positive shift takes place, no negative shift may occur, and vice versa. Fig. 8 at page 25 shows the data placement algorithm and the data movements associated with it.

## 3.4 The CREW Algorithm

Below we present the pseudo code for the implementation of the CREW version of Cole's parallel merge sort on a distributed memory architecture. The macro *MERGE_UP*() is defined at the end. The algorithm performs in $4 \times \log n$ parallel steps. The data movements of the algorithm can easily be followed step by step using Fig. 8.

---

*Parallel_Merge* ()
    $U_0^0$ = Initial Data.
    for ($i = 0$; $i < 3 \times \log n$; $i = i + 3$)
        For every $U_{i+1}^l \in W_{i+1}(U)$ do in parallel

            For $l = b(i+1)$
                Form the cover list $Z_{i+1}^l$ from $U_i^{l-1}$ (every 4th element)
                *MERGE_UP* $(U_{i+1}^l)$
                $+Shift\ (U_{i+1}^l)$

            For $l > b(i+1)$
                $Receive$ (+, $U_i^{l-1}$)
                Form the cover list $Z_{i+1}^l$ from $U_i^{l-1}$ (every 4th element)
                *MERGE_UP* $(U_{i+1}^l)$
                $+Shift\ (U_{i+1}^l)$

endparallel

For every $U_{i+2}^l \in W_{i+2}(U)$ do in parallel

For $l = b(i+2)$
Form the cover list $Z_{i+2}^l$ from $U_i^{l-1}$ (every 2th element)
*MERGE_UP* $(U_{i+2}^l)$

For $l > b(i+2)$
*Receive* (+, $U_{i+1}^{l-1}$)
Form the cover list $Z_{i+2}^l$ from $U_{i+1}^{l-1}$ (every 4th element)
*MERGE_UP* $(U_{i+2}^l)$
$-$*Shift* $(U_{i+2}^l)$
endparallel

For every $U_{i+3}^l \in W_{i+3}(U)$ do in parallel

/* Performed in two parallel steps */

/* First step: $n$ processors are used */
For $l = b(i+3)$
Form the cover list $Z_{i+3}^l$ from $U_i^{l-1}$ (every element)
*MERGE_UP* $(U_{i+3}^l)$

/* Second step */
For $l > b(i+3)$
Form the cover list $Z_{i+3}^l$ from $U_{i+2}^{l-1}$ (every 4th element)
if $|Z_{i+3}^l| > 1$ then
*Receive* (-, $U_{i+2}^l$)
endif
*MERGE_UP* $(U_{i+3}^l)$
$+$*Shift* $(U_{i+3}^l)$
endparallel
endfor
end

macro *MERGE_UP* $(U_i^l)$
/* Compute ranks $U_{i-1}^l \rightarrow X_i^l$ and $U_{i-1}^l \rightarrow Y_i^l$ */
*Compute_R4*($X_i^l$, $Y_i^l$, $U_{i-1}^l$)

/* Compute $U_i^l = X_i^l \cup Y_i^l$ */
if $|Z_i^l| = 1$ then

```
        Sort (U_i^l, X_i^l, Y_i^l)
    else
        Merge_with_help (U_i^l, X_i^l, Y_i^l, U_{i-1}^l)
    endif
end
```

---

The function *Sort* $(U_i^l,\ X_i^l,\ Y_i^l)$ performs a two-element sort involving two processors and leaves the result in $U_i^l$. The function *Merge_with_help* $(U_i^l,\ X_i^l,\ Y_i^l,\ U_{i-1}^l)$ defined below relies on Cole's algorithm to perform the merge in constant time; the algorithm was briefly described in Section 2. $X_i^l$ and $Y_i^l$ are two ordered lists and $U_{i-1}^l$ is the list resulting from the merge performed during the previous stage of the algorithm ($U_{i-1}^l = X_{i-1}^l \cup Y_{i-1}^l$). To perform the merge in constant time, the function assumes that the ranks $U_{i-1}^l \rightarrow X_i^l$ and $U_{i-1}^l \rightarrow Y_i^l$ have already been computed. At this point, some notation has to be defined ($L$ below stands for an $X$, $Y$ or $U$ list):

| | |
|---|---|
| $e \rightarrow L$ | stands for the rank of an element $e$ in the list $L$, |
| $*(e \rightarrow L)$ | stands for the value of an element in $L$ whose rank is given by $e \rightarrow L$, |
| $e.left$ | stands for the rank of an element $d$ where $d < e$ and $d$ and $e$ belong to the same list, |
| $e.right$ | stands for the rank of an element $f$ where $f > e$ and $f$ and $e$ belong to the same list. |

The merge reduces to compute the cross-ranks $X_i^l \leftrightarrow Y_i^l$. Recall that there is a processor $P_k$ assigned to the $k$th element of a list $L$. During the merge operation $U_i^l = X_i^l \cup Y_i^l$, the set of processors holding $U_i^l$ is the juxtaposition of the set of processors holding $X_i^l$ and the set of processors holding $Y_i^l$. Finally, let $P_j$ be the processor holding the first element of a list $L$ and$P_l$ be the processor holding the last element of $L$.

---

*Merge_with_help* $(L_i^l, M_i^m, N_i^l, SL_h^l)$

/* Compute $M_i^m \cup N_i^l$ with help of $SL_h^l$. The result is left in $L_i^l$. */

For every processor $P_k$ holding an element of $SL_h^l$ do in parallel

/* *R1 COMPUTATION* */

/* $P_k$ forms the interval $[e, f)$ */
$f$ = element of $SL_h^l$ held by $P_k$
$e$ = element of $SL_h^l$ held by $P_{k-1}$
/* The last statement implies transferring the element held by $P_k$ to $P_{k+1}$,
 * if $P_{k+1}$ exists.
 */

/* Remarks :
 * $P_j$ will handle the interval $[-\infty, f)$.
 * $P_l$ will handle the intervals $[e, f)$ and $[f, \infty)$ in two consecutive steps.
 */

/* Forming sets in $M_i^m$ */
/* Compute the rank of the leftmost element */
$x\_left\_rank = e \rightarrow M_i^m$
if $e > *(e \rightarrow M_i^m)$ then
    $x\_left\_rank = x\_left\_rank + 1$
endif
/* Compute the rank of the rightmost element */
$x\_right\_rank = f \rightarrow M_i^m$
if $f = *(f \rightarrow M_i^m)$ then
    $x\_right\_rank = x\_right\_rank - 1$
endif

/* Forming sets in $N_i^l$ */
/* Compute the rank of the leftmost element */
$y\_left\_rank = e \rightarrow N_i^l$
if $e > *(e \rightarrow N_i^l)$ then
    $y\_left\_rank = y\_left\_rank + 1$
endif
/* Compute the rank of the rightmost element */
$y\_right\_rank = f \rightarrow N_i^l$
if $f = *(f \rightarrow N_i^l)$ then
    $y\_right\_rank = y\_right\_rank - 1$
endif

if $x\_left\_rank > x\_right\_rank$ or $y\_left\_rank > y\_right\_rank$ then

    $\forall\ g\ \in M_i^m \mid x\_left\_rank \leq g \rightarrow M_i^m \leq x\_right\_rank$
        $g \rightarrow N_i^l = e \rightarrow N_i^l$

$\forall\ g\ \in N_i^l \mid y\_left\_rank \leq g \rightarrow N_i^l \leq y\_right\_rank$
$g \rightarrow M_i^m = e \rightarrow M_i^m$

else
/* Compute the cross-ranks between the set of elements in $M_i^m$
* and the set of elements in $N_i^l$
*/

*Compute_cross_ranks (x_left_rank, x_right_rank, y_left_rank, y_right_rank)*

/* The low-level function *Compute_cross_ranks (xlr, xrr, ylr, yrr)*, whose
* implementation is not shown, will relate the set of processors holding
* the elements of $M_i^m$ with rank $s$, where $xlr \leq s \leq xrr$, with the set of
* processors holding the elements of $N_i^l$ with rank $t$, where $ylr \leq t \leq yrr$,
* in order to compute the desired cross-ranks. This can be done in at most
* 5 comparisons.
*/
endif

For every processor $P_k$ holding an element $g$ of $M_i^m$ or $N_i^l$ do in parallel

/* *COMPUTE THE FINAL RANK* */

$g \rightarrow L_i^l = g \rightarrow M_i^m + g \rightarrow N_i^l$

/* *STORE DATA NEEDED TO COMPUTE R4* */

Remember whether $g$ is an element from $M_i^m$ or from $N_i^l$, next:
if $g \in M_i^m$
$g.left = *(g \rightarrow N_i^l) \rightarrow L_i^l$
$g.right = *(g \rightarrow N_i^l + 1) \rightarrow L_i^l$
endif

if $g \in N_i^l$
$g.left = *(g \rightarrow M_i^m) \rightarrow L_i^l$
$g.right = *(g \rightarrow M_i^m + 1) \rightarrow L_i^l$
endif

/* *PERFORM THE MERGE* */

/* This next statement implies transferring the element $g$ to the processor

```
        * holding the element of L_i^l with rank g → L_i^l.
        */

        *(g → L_i^l) = g
end

Compute_R4 (X_i^l, Y_i^l, U_h^l)
    /* Compute the ranks U_h^l → X_i^l and U_h^l → X_i^l.
     * It is assumed that the cross-ranks R2 and R3 have
     * already been deduced as explained in Section 2.
     */

    For every processor P_k holding an element e from U_h^l do in parallel

        /* Compute for elements in U_h^l came from X_{i-1}^l their ranks in Y_i^l */
        Compute_rank (e, (*e.left) → Y_i^l, (*e.right) → Y_i^l)

        /* Compute for elements in U_h^l came from Y_{i-1}^l their ranks in X_i^l */
        Compute_rank (e, (*e.left) → X_i^l, (*e.right) → X_i^l)

        /* Save the value *(e → X_i^l) (or *(e → Y_i^l))
         * which will be used when computing R1.
         */

        /* The low-level function Compute_rank (e, lr, rr), whose
         * implementation is not shown, will relate the processor P_k holding e with
         * the set of processors holding the elements from X_i^l (or Y_i^l) with
         * rank s, where lr ≤ s ≤ rr, in order to compute the rank
         * e → X_i^l (or e → Y_i^l). This can be done in at most 3 comparisons.
         */
end
```

---

The read conflict appears clearly when computing the rank $e \rightarrow Y_i^l$ (or $e \rightarrow X_i^l$) in the last function: the number of elements inside the rank interval [*e.left*, *e.right*] is not bounded. Let $S$ be the set of elements within this rank interval. A possible solution would be to perform a parallel prefix operation among the processors holding $S$, where the elements to be broadcast are those in $Y_i^l$ (or $X_i^l$) within the rank interval $(r, t]$ (at most 3 elements, see Fig. 5 at page 6). This may increase the parallel time complexity by a $\log n$ factor. In the next section, a more complex version of the algorithm which avoids read conflicts is considered.

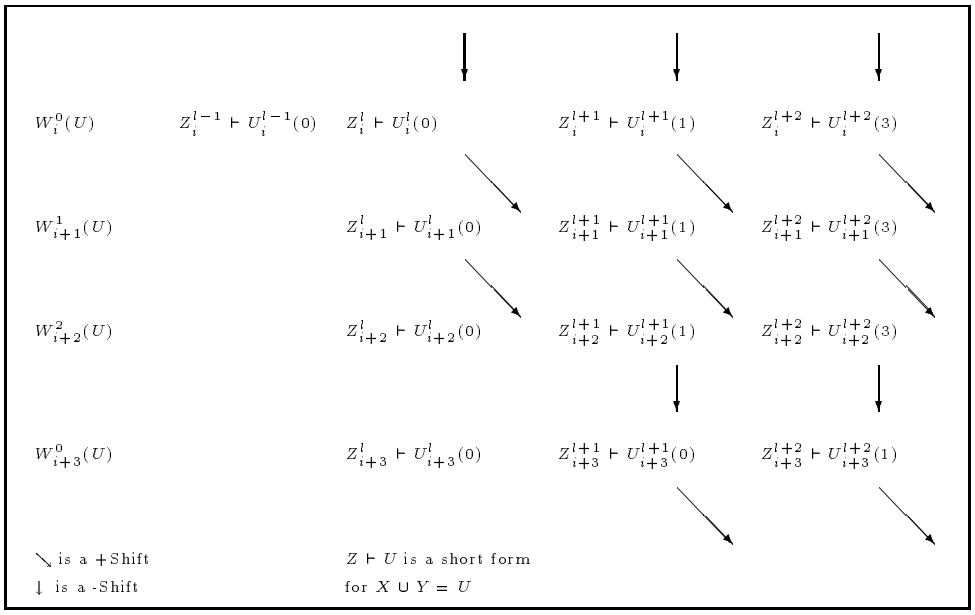



Figure 8: *The data placement algorithm and data movements. The number between parenthesis corresponds to the offset assigned by the algorithm.*

# 4 Cole's Parallel Merge Sort: the EREW algorithm

The EREW version of the parallel merge sort algorithm relies on the CREW version presented in Sections 2 and 3. Therefore, the analysis given so far is still valid. A brief description of the algorithm is presented in this section; for a complete description the reader is referred to [7]. In the EREW version, two additional lists are introduced in order to allow the merge to be performed in constant time and without read conflicts. First, let us give the following notation: $L$ $(node)$ refers to the list $L$ residing at (or coming from) node *node* in the binary tree. The new lists are $D_i^l$ and $SD_i^l$. $D_i^l(v)$ is the list resulting from the merge of $Z_i^{l+1}(w)$ and $SD_i^l(u)$, where the former is a cover list of $U_{i-1}^l(w)$ and the latter a cover list of $D_{i-1}^{l+1}(u)$ (every 4th element). Node $u$ is the parent of nodes $w$ and $v$. Hence,

$$D_i^l(v) = X_i^{l+1}(w) \cup SD_i^l(u) \text{ and}$$

$$D_i^l(w) = Y_i^{l+1}(v) \cup SD_i^l(u).$$

$D_i^l$ is computed at the same algorithm stage as $U_i^l$. It is important to note that the

cover list coming from $D_i^l$ is denoted as $SD_{i+1}^{l-1}$ since it is a list to be used at a lower level of the tree during the next algorithm stage.

A stage of the EREW algorithm is performed in 5 steps described below. Each step will work with a set of lists shown in Fig. 9, where $SM$ and $SN$ are cover lists of $M$ and $N$ respectively. The following ranks are assumed to be known from the previous algorithm stage:

*i)* $SM \rightarrow M, SN \rightarrow N,$

*ii)* $SM \leftrightarrow SN,$

*iii)* $SM \rightarrow N, SN \rightarrow M,$

$SL$ is the list issued from the merge of $SM$ and $SN$ whereas $L$ is the list resulting from the merge of $M$ and $N$. The function *Merge_with_help* $(L, M, N, SL)$ presented in Section 3.4 is used to compute the cross-ranks $M \leftrightarrow N$. The ranks $SL \rightarrow L$ are immediately computed.

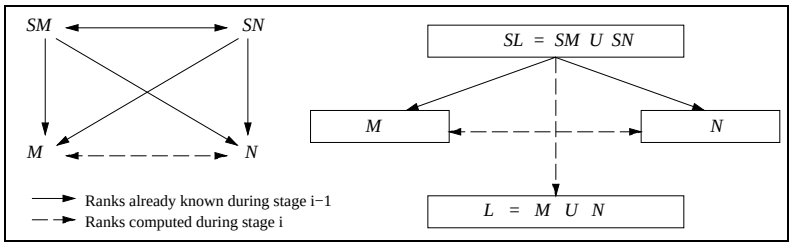



Figure 9: *EREW algorithm. Ranks computation during a step.*

At stage $i$ of the algorithm and at any node $v$ of level $l$, the following ranks are assumed to be known from stage $i-1$. Node $u$ is the father of nodes $v$ and $w$ and node $v$ is the father of nodes $x$ and $y$ (see Fig. 10 at page 28).

a) $Z_{i-1}^{l}(x) \leftrightarrow Z_{i-1}^{l}(y)$,

b) $Z_{i-1}^{l+1}(v) \rightarrow Z_{i}^{l+1}(v)$,

c) $Z_{i-1}^{l+1}(v) \leftrightarrow SD_{i-1}^{l}(u)$,

d) $SD_{i-1}^{l-1}(v) \rightarrow SD_{i}^{l-1}(v)$,

e) $Z_{i}^{l+1}(v) \leftrightarrow SD_{i}^{l-1}(v)$,

f) $U_{i-1}^{l}(v) \leftrightarrow SD_{i}^{l-1}(v)$,

g) $Z_{i}^{l+1}(v) \leftrightarrow D_{i-1}^{l}(v)$.

Since $D_{i-1}^{l}(v) = Z_{i-1}^{l+1}(w) \cup SD_{i-1}^{l}(u)$ and from g) $Z_{i}^{l+1}(v) \leftrightarrow D_{i-1}^{l}(v)$ is known, then the following ranks are deduced:

h) $Z_{i-1}^{l+1}(w) \rightarrow Z_{i}^{l+1}(v)$,

i) $SD_{i-1}^{l}(u) \rightarrow Z_{i}^{l+1}(v)$.

Similarly, since $U_{i-1}^{l}(v) = Z_{i-1}^{l}(x) \cup Z_{i-1}^{l}(y)$ and from f) we know $U_{i-1}^{l}(v) \leftrightarrow SD_{i}^{l-1}(v)$, then the ranks below are known:

j) $Z_{i-1}^{l}(x) \rightarrow SD_{i}^{l-1}(v)$, $Z_{i-1}^{l}(y) \rightarrow SD_{i}^{l-1}(v)$.

*Step 1.* To perform the merge $U_{i}^{l}(v) = Z_{i}^{l}(x) \cup Z_{i}^{l}(y)$, compute the cross-ranks $Z_{i}^{l}(x) \leftrightarrow Z_{i}^{l}(y)$. The ranks shown in Fig. 11 are available. Note that the ranks h and b in the figure are known respectively from h) and b) at any node of level $l-1$. This same step is also performed in the CREW algorithm presented in Section 2. Here, the ranks labeled with h (R4 in the CREW version) have already been computed without read conflicts during the previous algorithm stage. s$k$ in the figures stands for the ranks computed during step $k$. The following ranks are also computed.

$$U_{i-1}^{l}(v) \rightarrow U_{i}^{l}(v) \text{ and } Z_{i}^{l+1}(v) \rightarrow Z_{i+1}^{l+1}(v)$$

*Step 2.* To perform the merge $D_{i}^{l}(v) = Z_{i}^{l+1}(w) \cup SD_{i}^{l}(u)$, compute the cross-ranks $Z_{i}^{l+1}(w) \leftrightarrow SD_{i}^{l}(u)$. Fig. 12 pictures the ranks used to perform the merge. Note that the ranks j and d in the figure are known respectively from j) and d) at any node of level $l+1$. The cross-ranks are computed by means of the function *Merge_with_help*

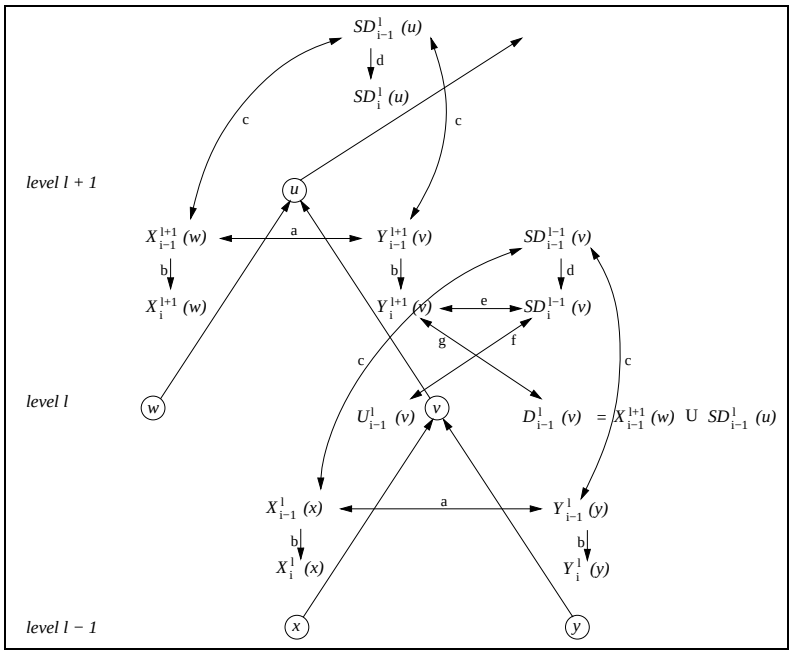



Figure 10: *a) EREW algorithm. Ranks known at stage i-1.*

$(D_i^l(v), Z_i^{l+1}(w), SD_i^l(u), D_{i-1}^l(v))$. The ranks below are also computed.

$$D_{i-1}^l(v) \rightarrow D_i^l(v) \text{ and } SD_i^{l-1}(v) \rightarrow SD_{i+1}^{l-1}(v).$$

*Step 3.* Compute the cross-ranks $Z_{i+1}^{l+1}(v) \leftrightarrow SD_{i+1}^{l-1}(v)$. The ranks that are needed are shown in Fig. 13. The rank s3(i) in the figure is deduced in the following way: since the cross-ranks $Z_i^{l+1}(v) \leftrightarrow Z_i^{l+1}(w)$ have been computed during step 1 at any node of level $l+1$ and the cross-ranks $Z_i^{l+1}(v) \leftrightarrow SD_i^l(u)$ have been computed during step 2, then the cross-ranks $Z_i^{l+1}(v) \leftrightarrow D_i^l(v)$, can easily be obtained (the rank of an element $e$ from $Z_i^{l+1}(v)$ in $D_i^l(v)$ is simply the sum of $e \rightarrow Z_i^{l+1}(w)$ and $e \rightarrow SD_i^l(u)$). Next, as $SD_{i+1}^{l-1}(v)$ is a cover list of $D_i^l(v)$, then the cross-ranks $Z_i^{l+1}(v) \leftrightarrow SD_{i+1}^{l-1}(v)$ are known.

Likewise, s3(ii) in Fig. 13 can be easily obtained: the cross-ranks $SD_i^{l-1}(v) \leftrightarrow Z_i^l(x)$ and $SD_i^{l-1}(v) \leftrightarrow Z_i^l(y)$ are known from step 2 at any node of level $l-1$. As above, $SD_i^{l-1}(v) \leftrightarrow U_i^l(v)$ is easily determined. Since $Z_{i+1}^{l+1}(v)$ is a cover list of $U_i^l(v)$, then the cross-ranks $SD_i^{l-1}(v) \leftrightarrow Z_{i+1}^{l+1}(v)$ are known.

*Step 4.* Compute the cross-ranks $U_i^l(v) \leftrightarrow SD_{i+1}^{l-1}(v)$. The ranks needed are shown in

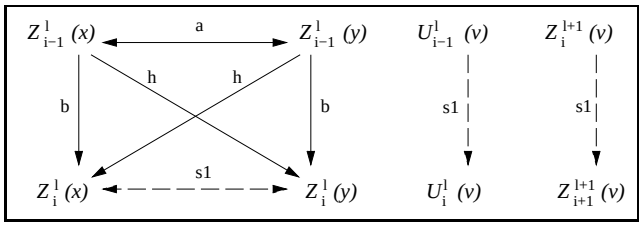



Figure 11: *EREW algorithm. Step 1.*

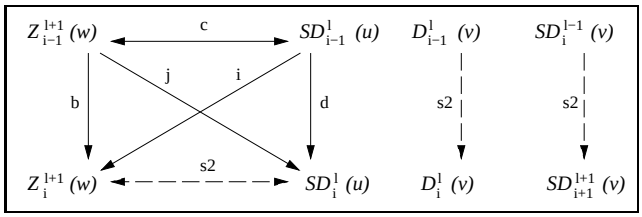



Figure 12: *EREW algorithm. Step 2.*

Fig. 14a).

*Step 5.* Compute the cross-ranks $Z^{l+1}_{i+1}(v) \leftrightarrow D^{l}_{i}(v)$. The ranks involved in the computation are shown in Fig. 14b).

# 5 Implementing the EREW Algorithm on a Distributed Memory Architecture

## 5.1 $D^l_i$ and $SD^l_i$ Data Placement

By making $D^l_i$ reside on the same set of processors holding $U^l_i$, the structure of the CREW algorithm remains unchanged. Moreover, the data movements applied to $U^l_i$ are also applied to $D^l_i$. $SD^l_i$ is a cover list of $D^{l+1}_{i-1}$ residing on the same set of processors holding $D^{l+1}_{i-1}$ (or $U^{l+1}_{i-1}$). Fig. 15 shows the active window for different values of $i$ and from level 7 to level 9 of an example of the EREW version ($N > 9$). There, we can mainly appreciate the data placement and the size of the new lists. They contain the same set of arguments as a $U^l_i$ list; their values are given below.

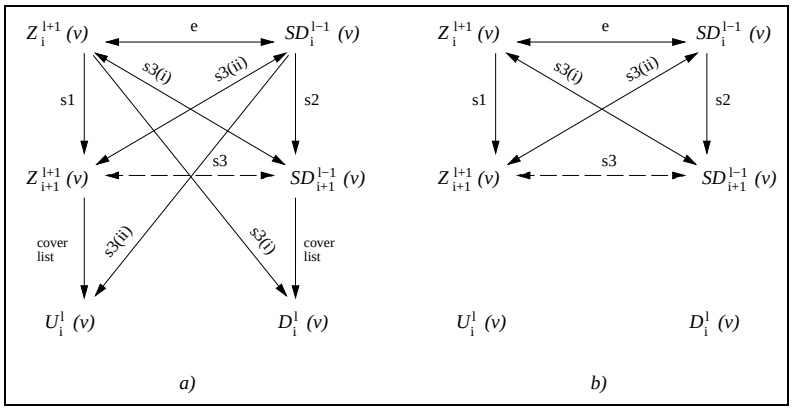



Figure 13: *EREW algorithm. a) Ranks used to compute steps 3, 4 and 5. b) Step 3.*

First, we start by the size of $D_i^l$, which is given by the sum of the sizes of the two lists involved in the merge operation.

$$|D_i^l| = |Z_i^{l+1}| + |SD_i^l|,$$

where $|SD_i^l| = \frac{|D_{i-1}^{l+1}|}{4}$, which implies a recurrence to the size of $D$ at the upper level during the previous algorithm stage. $|D_i^l| = 0$ for $l = N$. To replace the recurrence, the size of $D_i^l$ can be obtained by the formula given below.

$$|D_i^l| = \sum_{j=0}^{k} 2^{h-5j}$$

where $h = i - 2l - 3$ and $k = \min(\lfloor \frac{h}{5} \rfloor, N - 1 - l)$, then

$$|D_i^l| = \tfrac{1}{31} \times \left\{ 2^{h+5} - 2^{h-5k} \right\}.$$

Hence, the following upper bound on the size of $D_i^l$ is obtained:

$$|D_i^l| \leq \tfrac{32}{31} \times |Z_i^{l+1}| - \tfrac{1}{31} \tag{6}$$

As the value of $i$ progresses, the size of $D_i^l$ increments. Next, we will prove that for the maximum possible size of $D_i^l$, $|D_i^l| < |U_i^l|$. Recall that we make $D_i^l$ reside on the same set of processors holding $U_i^l$.

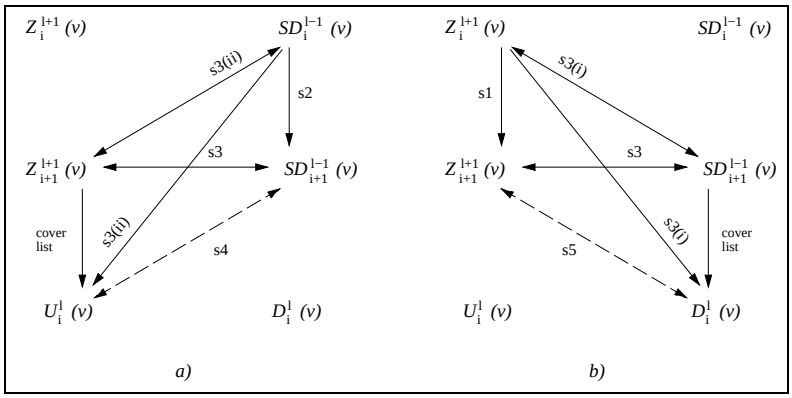



Figure 14: *EREW algorithm. a) Step 4. b) Step 5.*

The size of $D_i^l$ attains its maximum value when $i$ is equal to $3l+2$. This comes from the fact that the EREW algorithm does not compute $D_i^l$ for a value of $i$ equal to $3l+3$ (see Fig. 15). For $|U_i^l|$ the maximum value of $i$ is *end_level*; therefore, in order to guarantee the correct data placement for $D_i^l$ we need to prove the following relation:

$$|D_{3l+2}^l| < |U_{3l}^l|.$$

Using Eq. (6) to determine the maximum value of $|D_{3l+2}^l|$ we obtain:

$$\frac{32}{31} \times 2^{l-1} - \frac{1}{31} < 2^l$$
$$2^{l-0.95} - \frac{1}{31} < 2^l. \square$$

The rest of the arguments are known straightforwardly:

$s(D_i^l) = s(U_i^l)$,
$s(SD_i^l) = s(U_{i-1}^{l+1})$,

$d(D_i^l) = d(U_i^l)$,
$d(SD_i^l) = d(U_{i-1}^{l+1})$,

$offset(D_i^l) = offset(U_i^l)$,
$offset(SD_i^l) = offset(U_{i-1}^{l+1})$.

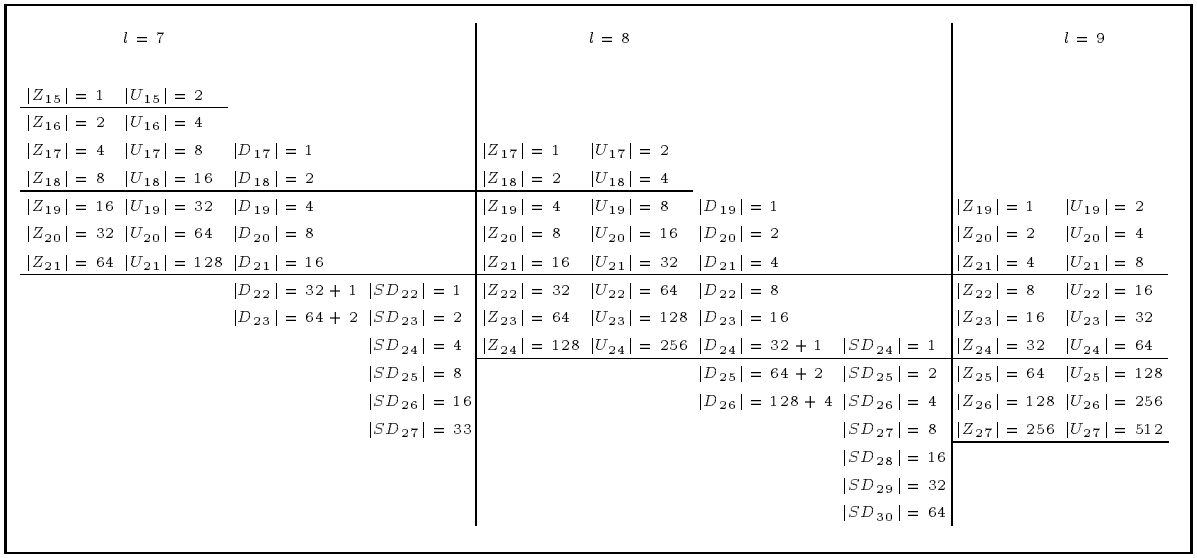



Figure 15: *Cole's Parallel Merge Sort: EREW version for $N > 9$.*

## 5.2 The EREW Algorithm

Below we present the pseudo code for the distributed memory implementation of the EREW version of Cole's parallel merge sort. The macros *MERGE_UP*() and *MERGE_DOWN*() are defined at the end. The function *Merge_with_help*() has been defined in Section 3.4 (the code related to R4 computation should be omitted). The algorithm performs in $4 \times \log n$ parallel steps using exactly $n$ processors.

---

*Parallel_Merge* ()
  $U_0^0$ = Initial Data.
  for ($i = 0$; $i < 3 \times \log n$; $i = i + 3$)
    for ($j = 1$; $j \leq 3$; $j = j + 1$)

      For every $U_{i+j}^l \in W_{i+j}(U)$ do in parallel

        For $l = b(i + j)$
          Form the cover list $Z_{i+j}^l$ from $U_i^{l-1}$ (every $\lfloor \frac{4}{j} \rfloor$th element)
          *Step 3, Step 4, Step 5*
          *Step 1*
          *MERGE_UP* $(U_{i+j}^l)$

```
                    if (j ≠ 3) then
                        Step 2
                        MERGE_DOWN (D^{l-1}_{i+j})
                        if (j = 1) then
                            +Shift (U^l_{i+j})
                            +Shift (D^l_{i+j})
                        endif
                    endif

                For l > b(i + j)
                    if (j ≠ 3) then
                        Receive (+, U^{l-1}_{i+j-1})
                        Receive (+, D^{l-1}_{i+j-1})
                        Form the cover list Z^l_{i+j} from U^{l-1}_{i+j-1} (every 4th element)
                    else
                        Form the cover list Z^l_{i+j} from U^{l-1}_{i+j-1} (every 4th element)
                        if |Z^l_{i+j}| > 1 then
                            Receive (-, U^l_{i+j-1})
                        endif
                        if |Z^l_{i+j}| ≥ 32 then
                            Receive (-, D^l_{i+j-1})
                        endif
                    endif
                    Step 3, Step 4, Step 5
                    Step 1
                    MERGE_UP (U^l_{i+j})
                    Step 2
                    MERGE_DOWN (D^{l-1}_{i+j})
                    if (j ≠ 2) then
                        +Shift (U^l_{i+j})
                        +Shift (D^l_{i+j})
                    else
                        −Shift (U^l_{i+j})
                        if (|U^l_{i+j}| ≤ 32) then
                            −Shift (D^l_{i+j})
                        endif
                    endif
            endparallel

        endfor
    endfor
end
```

```
macro MERGE_UP (U_i^l)
    /* Compute U_i^l = X_i^l ∪ Y_i^l */
    if |Z_i^l| = 1 then
        Sort (U_i^l, X_i^l, Y_i^l)
    else
        Merge_with_help (U_i^l, X_i^l, Y_i^l, U_{i-1}^l)
    endif
end

macro MERGE_DOWN (D_i^l)
    /* Compute D_i^l = Z_i^{l+1} ∪ SD_i^l */
    if |Z_i^{l+1}| ≥ 32 then
        Merge_with_help (D_i^l, Z_i^{l+1}, SD_i^l, D_{i-1}^l)
        /* Remark: The result of the merge operation is transferred
         * to the set of processors holding D_i^l. At the same time,
         * D_i^{l+1} is received from the upper level.
         */
    else
        D_i^l = Z_i^{l+1}
    endif
end
```

# 6 Conclusions

We gave a proposition to implement Cole's parallel merge sort on a distributed memory architecture. Both, the CREW and the EREW algorithms have been considered. A data placement algorithm has been presented as well as the associated data movements. Our proposition sorts $n$ items using exactly $n$ processors in $O(\log n)$ parallel time. The constant in the running time is only one greater than the one obtained for the PRAM model.

We envisage to consider the case where $n > p$, $n$ being the number of elements to sort and $p$ the number of processors employed. A first approach would be to consider that each of the $p$ processors contains $n/p$ virtual processors, and then to apply the actual algorithm taking into account that some data exchange will correspond to internal

read/write operations. A second approach consists in optimizing the previous one by sorting the elements locally at each processor using the best known sequential sort, next considering the $n/p$ elements in a processor as a "data unit" and finally applying the actual algorithm using $p$ processors and $p$ "data units".

# References


[1] Ajtai, M., Komlós, J. and Szemerédi, E. Sorting in $c \log n$ Parallel Steps. *Combinatorica*, 1983, Vol. 3, No. 1, p. 1-19.

[2] Batcher, K.E. Sorting Networks and their Applications. In : *Proceedings of the AFIPS Spring Joint Computer Conference*, 1968. Vol. 32, p. 307-314.

[3] Blelloch, G.E. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, November 1989, Vol. C-38, No. 11, p. 1526-1538.

[4] Blelloch, G.E. Prefix Sums and their Applications. Pittsburgh (PA) : School of Computer Science, Carnegie Mellon University, November 1990. Technical Report CMU-CS-90-190.

[5] Blelloch, G.E. *Vector Models for Data-Parallel Computing.* Cambridge (MA) : the MIT Press, 1990. 255 p.

[6] Blelloch, G.E., Plaxton, C.G., Leiserson, Ch.E., Smith, S.J., Maggs, B.M., Zagha, M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. Thinking Machines Corporation. 30 p.

[7] Cole, R. Parallel Merge Sort. *SIAM Journal on Computing*, August 1988, Vol. 17, No. 4, p. 770-785.

[8] Manzini, G. Radix Sort on the Hypercube. *Information Processing Letters*, 1991, No. 38, p. 77-81.

[9] Reif, J.H. (Ed.). *Synthesis of Parallel Algorithms.* San Mateo (CA) : Morgan Kaufmann Publishers, 1993. 1011 p.

[10] Reif, J.H. and Valiant, L.G. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, January 1987, Vol. 34, No. 1, p. 60-76.