A Lower Bound for Linear Approximate Compaction

S. Chaudhuri

MPI–I–93–146 Oktober 1993

A Lower Bound for Linear Approximate Compaction

S. Chaudhuri

MPI–I–93–146 Oktober 1993

A Lower Bound for Linear Approximate Compaction

Shiva Chaudhuri
Max-Planck-Institut für Informatik
Im Stadtwald
6600 Saarbrücken
Germany
E-mail: shiva@mpi-sb.mpg.de

Abstract

The λ-approximate compaction *problem is: given an input array of* n *values, each either 0 or 1, place each value in an output array so that all the 1's are in the first* $(1+\lambda)k$ *array locations, where* k *is the number of 1's in the input.* λ *is an accuracy parameter. This problem is of fundamental importance in parallel computation because of its applications to processor allocation and approximate counting. When* λ *is a constant, the problem is called* Linear Approximate Compaction *(LAC). On the CRCW PRAM model, there is an algorithm that solves approximate compaction in* $\mathcal{O}((\log\log n)^3)$ *time for* $\lambda = \frac{1}{\log\log n}$*, using* $\frac{n}{(\log\log n)^3}$ *processors. Our main result shows that this is close to the best possible. Specifically, we prove that LAC requires* $\Omega(\log\log n)$ *time using* $\mathcal{O}(n)$ *processors. We also give a tradeoff between* λ *and the processing time. For* $\epsilon < 1$*, and* $\lambda = n^{\epsilon}$*, the time required is* $\Omega(\log\frac{1}{\epsilon})$*.*

1 Introduction

A universal paradigm in parallel computing is the method of breaking up a task at hand into a number of smaller subtasks and doing the subtasks in parallel. Once the subtasks have been done, the results can be combined to accomplish the original task. A situation often encountered by algorithm designers is that for any instance of the original task, most of the arising subtasks can be done quickly, only a few are hard. An effective technique in this situation is to solve the easy subtasks quickly, then reallocate all the processors to the few remaining subtasks, making a substantial number of processors available to work on each subtask. Even though these subtasks are hard, it is possible to use the processor advantage to finish them quickly.

The efficiency of this method hinges entirely on how well the processors can be reallocated, making processor allocation a problem of fundamental importance. In most algorithms, the reallocation of processors is done by solving the *compaction* problem, defined as follows:
Compaction:
Input: $a_1, \ldots, a_n,\ a_i \in \{0,1\}$

Output: $b_1, \ldots, b_n$ such that if the input had k 1's, then for $1 \leq j \leq k$, $b_j = 1$ and for $j > k$, $b_j = 0$.

This problem is applied to processor allocation in the obvious way; the 1's in the input represent the subtasks that are hard. Once they have been compacted into the initial array positions, the available processors may be distributed evenly among them.

By solving compaction, one may also count the number of 1's in the input. This yields applications of compaction to counting problems, computation of threshold functions and computing parity. Thus, compaction has been extensively studied in parallel computation, especially in the context of CRCW PRAMs [19, 7, 17, 16]. Cole and Vishkin give an algorithm to solve compaction in $\mathcal{O}(\log n / \log\log n)$ time using an optimal number of processors on the ARBITRARY model [6]. On the other hand, MAJORITY reduces to compaction. Hence, compaction requires $\Omega(\log n/\log\log n)$ time, using a polynomial number of processors on the PRIORITY model. This follows from the lower bound of Beame and Håstad [1]. Thus it would seem that this method cannot yield better results for the problems it is applied to. (Henceforth, unless otherwise mentioned, all the upper bounds are for the ARBITRARY model and all lower bounds for the PRIORITY model. See [13] for descriptions of models of CRCW PRAMs.)

However, it has been observed that for the application of processor allocation, it is sufficient to solve compaction *approximately* [15, 10]. The same is often true when one is interested in counting the number of 1's [12, 4]. This leads to the following definition:

λ-Approximate Compaction:

Input: $a_1, \ldots, a_n,\ a_i \in \{0, 1\}$

Output: $b_1, \ldots, b_n$ such that if the input had k 1's, then for some $S \subseteq \{1, \ldots, \min\{(1+\lambda)k, n\}\}$, with $|S| = k$, $b_j = 1 \iff j \in S$. *(In other words, all the 1's in the input are compacted into k distinct locations among the first $(1+\lambda)k$ locations.)*

Once again, the application to processor allocation is obvious; the available processors are distributed evenly among the first $(1+\lambda)k$ array locations. If λ is a constant, for example, the number of processors allocated to each non-zero location is within a constant factor of the optimal number. The parameter λ is called the *padding factor*. For constant λ, the problem is called *Linear Approximate Compaction* (LAC).

Matias and Vishkin gave randomized algorithms to solve LAC in $\mathcal{O}(\log^* n)$ time with n processors [15]. Mackenzie gave an $\Omega(\log^* n)$ lower bound for this problem [14]. Thus the randomized complexity of LAC is known.

In the deterministic case, until recently, no algorithms that ran in $o(\log n/\log\log n)$ time were known. The only lower bounds known were for very small values of λ, i.e. $\lambda \leq n^{-\epsilon}$ [20]. These bounds can be obtained by a reduction from MAJORITY. However, this method does not yield good bounds for larger values of λ. For instance, when $\lambda \geq \frac{1}{\log\log n}$, there is no obvious lower bound for it. Recently, Hagerup gave an algorithm that solves this problem in $O((\log\log n)^3)$ time for $\lambda = \frac{1}{\log\log n}$, using $\frac{n}{(\log\log n)^3}$ processors [10]. This shows that it is strictly easier than exact compaction. In this paper, we show that this result is close to the best possible. Specifically, we prove that LAC requires $\Omega(\log\log n)$ time using $\mathcal{O}(n)$ processors. The lower bound is robust, in that it holds for nonuniform algorithms as well. It has been observed that there is an n-processor nonuniform algorithm for LAC that runs in time $\mathcal{O}(\log\log n)$ [11]. To improve the lower bound, therefore, requires explicit use of the uniformity of the algorithm.

We actually prove a general tradeoff between λ and the time required. It is natural to expect that the smaller the padding factor, the harder it is to solve the problem. This is reflected in the tradeoff. For $\lambda \geq 1$, we show that λ-approximate compaction requires $\Omega(\log(\frac{\log n}{\log(1+\lambda)}))$ time. Ragde showed that for $\lambda = k^3$, λ-Approximate Compaction can be solved in constant time with n processors [17]. Hagerup extended this by giving a constant time algorithm for $\lambda = k^\epsilon$ for any $\epsilon > 0$, where the constant depends upon ϵ [8]. Our tradeoff shows that when $\epsilon < 1$ and $\lambda = n^\epsilon$, the time required is $\Omega(\log \frac{1}{\epsilon})$, for sufficiently large n. Thus as $\epsilon \to 0$, the time required grows to infinity.

The techniques used in this paper are based on the small-domain lower bound methods of Chaudhuri and Radhakrishnan [5]. These methods are quite general. A large portion of the proof is applicable to any PRAM algorithm, and only a small portion utilizes the properties special to LAC. Thus, the techniques used may be of independent interest.

2 Preliminaries

2.1 The Model

In this section we give a detailed definition of the lower bound model. The reader willing to make commonsense assumptions about the model is encouraged to use this section as a reference to clarify issues that may arise when studying the proofs.

The model under consideration is sometimes referred to as the "Ideal" or "Full-information" PRAM. It has $P = P(n)$ processors $p_1, \ldots, p_P$, and a shared memory consisting of an infinite number of cells. Each cell is capable of holding words of arbitrary length. Each processor has a private memory of infinite size. Initially, the private memory is blank. Processors are allowed to simultaneously read and write memory cells; write conflicts are resolved using the PRIORITY rule [13]. Under this rule, when several processors simultaneously write to a cell, the processor with the smallest index succeeds in writing to the cell. Each processor has a *program*, i.e. a sequence of instructions that it executes. The program may vary depending on the index of the processor and on the problem size, n. We do not require the program to be parameterized by n, thus nonuniform algorithms are permitted.

Initially the input is assumed to be in the first n cells of shared memory, the output is to be written into these n cells.

The machine operates in synchronous steps, each step consisting of a *read* phase, a *compute* phase and a *write* phase. During the read phase, each processor may select a cell and read its contents. During the compute phase each processor may perform an arbitrary computation, referring an arbitrary number of times to its private memory. The only restriction is that it is not permitted to access the shared memory. During the write phase, each processor may select a cell and write to it.

The time taken by a computation is defined to be the number of steps taken during the computation.

The action of a processor at any given time is completely determined by the history of the

processor through the computation. Since a processor may perform any computation in the compute phase, we may as well assume that it keeps a complete record of whatever it reads in the shared memory in its private memory. This allows us to make the following definition:
Definition: The *state* of a processor, at time t is defined to be its program, and everything that is written in its private memory at time t.
We may now assume that the action of a processor at time t depends only on its state at time t. For any computation, each processor has exactly one initial state (since its private memory has nothing written in it.

Since there is no restriction on the wordsize or on the compute phase, whenever a processor writes, it may as well write the entire history of its computation; hence the name Full-information PRAM. Lower bounds on this model depend crucially on limiting the amount of information that processors can communicate to each other through the shared memory. Proving lower bounds on this model gives insight into the intrinsic difficulty of solving problems in parallel.

2.2 Partial Inputs and the Computation Graph

In the following, A will be an algorithm solving linear approximate compaction. For inputs of size n, let A use $P(n)$ processors and take $k(n)$ steps. We will use P and k for $P(n)$ and $k(n)$ respectively, from now on.

A *partial input* is an element of $\{0,1,*\}^n$. For a partial input b, we denote by $X(b)$ the set of inputs consistent with b. That is, $X(b) = \{x \in \{0,1\}^n : \mathit{for}\ i = 1,\ldots,n,\ b_i \neq * \rightarrow b_i = x_i\}$. For partial inputs a and b, we say a is a refinement of b, and write $a \leq b$, if $X(a) \subseteq X(b)$.

For a given partial input b, consider a processor p at time t. The set of inputs consistent with b defines a set of states that p may be in, on inputs consistent with b. This set of states, in turn, defines the possible actions of p at time t, in particular, it defines the set of memory locations that p may read from, or write to. This gives us a way to identify (and consequently, limit) the amount of information that p may read from, or write to, the shared memory. We formalize this by modelling the computation of A on a graph. Let b be a partial input of size n. The *computation graph* of A on b, $G(b)$, is defined as follows.

$$V(G(b)) = \{(c,i) : c\ \mathit{is\ a\ cell\ of\ memory\ and}\ 0 \leq i \leq k\}.$$

That is, we have $(k+1)$ levels; in each level we have one vertex for each cell in the memory. The set of vertices in level i will be called V_i. The directed edges go from vertices at one level to the vertices of the next level. Every edge is labelled by a processor. $E(G(b))$ contains the edge $((c,i),(d,i+1))$ labelled p if on some input in $X(b)$, processor p reads cell c and writes to cell d in step $(i+1)$. We use $f_v(b)$ to denote the indegree of vertex v in the graph $G(b)$. Initially, bit i of the input is assumed to be in cell i; finally, bit i of the output is assumed to be in cell i. We refer to vertex $(i,0)$ as α_i (the *input* vertices) and vertex (i,k) as β_i (the *output* vertices).

Let $a \in \{0,1\}^n$. We shall associate with each vertex of $G(a)$ a content. The content associated with (c,i) is the content of the cell c after step i (that is, just before the write of step $(i+1)$ changes it) in the computation of A on the input a. We call this content $\mathit{content}(a,(c,i))$. Similarly, for a processor p and an input $a \in \{0,1\}^n$, $\mathit{state}(a,(p,i))$ is the state of processor p

just before the write of step $(i+1)$ in the computation of A on input a. For a partial input b, let

$$\begin{aligned} contents(b,(c,i)) &= \{content(x,(c,i)) : x \in X(b)\}; \\ states(b,(p,j)) &= \{state(x,(p,i)) : x \in X(b)\}. \end{aligned}$$

We say that (c,i) is a *fixed* vertex if $|contents(b,(c,i))| = 1$; otherwise we say (c,i) is a *free* vertex. Note that the above definitions depend on the algorithm A and the size of input n. These parameters will be clear from the context where they are used.

We model the computation of the algorithm A on the computation graph as follows. We say that a processor p reads from cell (c,i) and writes to cell $(d,i+1)$ when we mean that in the step $(i+1)$ of the computation of the algorithm A, p reads cell c and writes to cell d.

3 The Lower Bound

3.1 Overview of the method

Let b be a partial input in which at most $n/8$ input positions have a value of 0 or 1. Consider a vertex (c,i) in the computation graph of algorithm A on partial input b. Let y be the content of (c,i) on some partial input consistent with b, that it, y is a possible content for (c,i) in the computation graph. Let $S_i(y)$ be a set of input variables with the following properties: (i) These variables have value $*$ in b, and (ii) there is a value (0 or 1) for each variable in the set such that when we set all variables to their corresponding values, (c,i) is fixed to content y. That is, when we set the values and consider the computation graph of A on the resulting partial input, (c,i) is a fixed vertex with content y.

Under certain circumstances, this can give us a contradiction. To see this, consider the following example. Let b be a partial input with at most $n/8$ input co-ordinates having 0-1 values. Let A be an algorithm for 1-Approximate Compaction. Suppose we can find a vertex (c,i) such that (c,i) is output vertex β_j and $n/2+1 \leq j \leq n$, and 1 is a possible content for (c,i) on partial input b. Suppose also that $|S_i(1)| \leq n/8$. Then, as above, we can find a partial input b' with at most $n/4$ 1's, such that output vertex β_j has content 1 on all inputs consistent with b. In particular, the input obtained by setting all the $*$'s in b' to 0 has at most $n/4$ 1's, yet, in the output, β_j will have value 1, contradicting the claim of 1-approximate compaction.

Our proof uses essentially the ideas outlined above. The only problems are that we may not be able to find a suitable β_j and if we do, then $|S_i(1)| \leq n/8$ may not hold. We are able to surmount these problems as follows. First, given any algorithm A, we obtain a partial input b such that the computation graph has the property that for any (c,i), the corresponding S_i has "small" cardinality. Intuitively, we may expect that as i grows, so does S_i. The main part of our analysis is showing that the growth of S_i can be bounded. This is called the *preprocessing* phase and is described in Section 3.2. The partial input b is carefully chosen so that the number of 0-1 co-ordinates is very small. This takes care of the second problem. We then show that if the number of 0-1 co-ordinates in a partial input is small, there exists an output vertex β_j with the desired properties. After this, we are in a situation like the example above. The final argument (in Theorem 3.1) finds a partial input and an output vertex (c,i) with content fixed to 1 that are inconsistent with each other if i is too small. We thereby conclude that the algorithm cannot finish in less than i steps.

3.2 Preprocessing

Let the algorithm A and the computation graph be defined as in Section 2.2. Define $d_1 = m \geq 20$, where m is a parameter whose value we determine later, and define $d_{i+1} = d_i^6$. Then $d_i = m^{6^{i-1}}$. Notice that we shall use the facts that $d_i \geq 20$ and $d_{i+1} \geq d_i^6$ in making several estimates later.

Definition: For a given i, $1 \leq i \leq k$, say b is a *level-i bounding* partial input if, in $G(b)$, free vertices at level j have indegree less than d_j for $1 \leq j \leq i$.

Note that the above definition implies that if b is a level-i bounding partial input, then it is also a level-j bounding partial input for $j \leq i$. Also, if $b' \leq b$, then b' is also a level-i bounding input.

Let b be a level-i bounding partial input and let $x \in X(b)$. Consider the state of a processor p after the read of step $(i+1)$. We wish to refine b so that this state is fixed at $state(x,(p,i))$. More precisely we wish to obtain b' so that $b' \leq b$ and $states(b',(p,i)) = \{state(x,(p,i))\}$. Fixing the content of a cell is defined similarly i.e. we want $b' \leq b$ and $contents(b',(c,i)) = \{content(x,(c,i))\}$. In Lemma 3.1 we give procedures to find partial inputs that fix processors and cells to the state they have on any chosen input x. Furthermore, the procedures ensure that the partial inputs found are consistent with x.

The state of p after the read of step $i+1$ is completely determined by its state after the ith read, and the contents of the cell it reads in step $i+1$. Note that if the state of p after the ith read is fixed, then it always reads the same cell in the $i+1$th read. Fixing the contents of this cell ensures that the state of p after the $i+1$th read is fixed.

Similarly the content of a cell c after the $i+1$th write is completely determined by the states of the processors that can write to c in the $i+1$th write, and the content of c after the ith write. If the states of these processors are fixed, then so is the content of c, except in the case when no processor writes to c in the $i+1$th write. In either case, also fixing the content of c after the ith write ensures that the content of c after the $i+1$th write is fixed. This method of fixing processors and cells is formalized in

Lemma 3.1 *Let b be a level-i bounding partial input and let $x \in X(b)$. For any given processor p (cell c), there exists a partial input b' such that $x \leq b' \leq b$ and* $\text{states}(b',(p,i)) = \{\text{state}(x,(p,i))\}$ ($\text{contents}(b',(c,i)) = \{\text{content}(x,(c,i))\}$). *Further, the number of $*$'s in b that have a value of 0 or 1 in b' is at most d_i^2.*

Proof. We show the existence of b' by describing procedures that find such a partial input. The procedures are recursive and defined below.

FixProc$(x,b,(p,i))$: If $i=0$, we know that p reads the same input position in the first step no matter which input is presented to the algorithm. To fix the state of p after the first read, we fix that input bit of b (if it is not already set) consistently with x. The resulting partial input is our b'.

For $i > 0$, let b'' be the partial input produced by $\mathit{FixProc}(x, b, (p, i-1))$. Then, for each input in $X(b'')$, p reads the same cell, (c, i) say, in the read of step $(i+1)$. We use the $\mathit{FixCell}(x, b'', (c, i))$ to fix the contents of this cell and call the resulting partial input b'.

$\mathit{FixCell}(x, b, (c, i))$: If $i = 0$, then (c, i) is an input bit. We set this bit consistently with x and the resulting partial input is our b'.

If $i > 0$, let $p_1, p_2, \ldots, p_t$ be the processors that write to cell (c, i). We now define partial inputs $b(0), b(1), \ldots, b(t)$ inductively. Let $b(0) = b$, and for $j = 1, \ldots, t$, $b(j) =$ output of $\mathit{FixProc}(x, b(j-1), (p_j, i-1))$. To get the final partial input b', we invoke $\mathit{FixCell}(x, b(t), (c, i-1))$.

Notice that whenever $\mathit{FixCell}(x, b^*, (c^*, j))$ is invoked either directly, or recursively, we have $b^* \leq b$ and $j \leq i$. Since b is a level-i degree bounding input, it is also a level-j bounding input, and so is b^*. Thus, the indegree of (c^*, j) in $G(b^*)$ will be at most $d_j - 1$. Hence, in the preceding paragraph, $t \leq d_i - 1$.

It is easy to check using induction that the procedures fix correctly, i.e. the considered processor or cell is indeed fixed on b' to the same state or content that it has on input x. Observe that b' is obtained by setting $*$'s in b to 0 or 1, and the setting is always consistent with x. This ensures that $x \leq b' \leq b$. It remains to show that at most $d_i{}^2$ $*$'s in b are set to 0 or 1 in this process.

Let $N_i(p)$ be the number of $*$'s in b that are set to 0 or 1 by $\mathit{FixProc}(x, b, (p, i))$. Let $N_i = \max_p N_i(p)$. Similarly, let $M_i(c)$ be the number of $*$'s in b that are set to 0 or 1 by $\mathit{FixCell}(x, b, (c, i))$. Let $M_i = \max_c M_i(c)$. It is easy to check that the following inequalities hold for $i \geq 1$.

$$\begin{aligned} M_i &\leq M_{i-1} + (d_i - 1)N_{i-1}; \\ N_i &\leq N_{i-1} + M_i. \end{aligned}$$

From these, with $N_0 = 1$, $M_0 = 1$ and the stated bounds on the values of d_j, the following bounds can be deduced, for $i \geq 1$.

$$M_i \leq 2^{i-1}(d_1 \ldots d_i) \leq d_i^2; \tag{1}$$

$$N_i \leq 2^i(d_1 \ldots d_i) \leq d_i^2. \tag{2}$$

∎

We shall now analyze the computation graph of algorithm A. We shall find a level-k bounding partial input b. That is, in the graph $G(b)$, the indegree of a *free* vertex at level i will be less than d_i, for $i = 1, 2, \ldots, k$. We next show how such a partial input with a small number of 0's and 1's can be obtained.

The partial input b is produced in stages. The intermediate partial inputs produced will be called $b^0, b^1, \ldots, b^k$. In the end we shall set $b = b^k$. Initially, we set $b^0 = *^n$. Now, in the graph $G(b^0)$, there may be free vertices in level 1 that have degree d_1 or higher. In *STAGE 1* of our procedure, we refine b^0 to obtain b^1. In $G(b^1)$ the degree of every free vertex at level 1 will be less than d_1.

When we come to *STAGE* i, we already have a partial input b^{i-1} such that, in $G(b^{i-1})$, every free vertex at level j, $j = 1, 2, \ldots, i-1$, has degree less than d_j. Our task in *STAGE* i is to ensure that this holds for level i vertices also. We obtain a refinement b^i of b^{i-1} so that, in $G(b^i)$, every free vertex at level i has degree less than d_i. Note that the degree of any vertex cannot increase upon refinement.

We now describe the processing done at *STAGE* i. Consider the graph $G(b^{i-1})$. A free vertex v at level i will be called a *high degree* vertex if $f_v \geq d_i$. To obtain b^i we fix high degree vertices as described below. Define $b^* = b^{i-1}$.

High Degree Vertices. For each high degree vertex v we do the following: Let the highest-priority processor writing to v be p. There is some input $x \in X(b^*)$ on which p writes to v. We fix the state of p to the state it has on input x using $FixProc(x, b^*, (p, i-1))$. Redefine b^* to be the new partial input. On every input in $X(b^*)$, p writes to v and, being the highest priority processor, always succeeds. To fix the state of p we need set at most N_{i-1} inputs.

At the end of this process, all the free vertices at level i have indegree less than d_i.

This completes the description of the processing at *STAGE* i.

Upon application of the above process to levels $1, \ldots, k$, we have a level-k bounding partial input. We next show that for such an input, the number of possible states and contents at a given level is bounded.

Let b be a level-i bounding input. With respect to $G(b)$ define $Y_i(b)$ and $Z_i(b)$ as follows.

$$\begin{aligned} Y_i(b) &= \max\{|contents(b,(c,i))| : c \textit{ is a memory cell}\}; \\ Z_i(b) &= \max\{|states(b,(p,i))| : p \textit{ is a processor}\}. \end{aligned}$$

Lemma 3.2 *Let b be a level-i bounding partial input. Then $Z_i(b) \leq 2^{2^i} \prod_{j=1}^{i} d_j^{2^{i-j}}$.*

Proof. We drop the parameter b in the notation. We have that $Y_0 = 2$ and $Z_0 = 2$, since, initially, each cell has 0 or 1 and each processor, after the first read, can be in at most 2 states.

Consider a vertex (c,i) $(i > 0)$ in the graph $G(b)$. Let $d < d_i$ be the indegree of (c,i). Let $p_1, \ldots, p_d$ be the processors that label the d edges. Let the number of states in which processor p_j writes to (c,i) be S_j. The content of (c,i) is determined by the state of the processor that succeeds in writing to (c,i), or, if no processor writes to (c,i), by the content of $(c,i-1)$. Thus, we have

$$|contents(b,(c,i))| \leq \sum_{j=1}^{d} S_j + |contents(b,(c,i-1))|.$$

By induction, $S_j \leq Z_{i-1}, \forall j$ and $|contents(b,(c,i-1))| \leq Y_{i-1}$. Thus, for $i \geq 1$,

$$Y_i \leq (d_i - 1)Z_{i-1} + Y_{i-1}$$

The number of states of a processor after the ith read is at most the product of the number of states it had after the $i-1$th read and the number of contents of the cell it read at the ith read. Thus

$$Z_i \leq Z_{i-1} Y_i.$$

Let $\Delta_i = 2^{2^i} \prod_{j=1}^{i} d_j^{2^{i-j}}$. It can then be shown by induction on i that $Z_i \leq \Delta_i$ and $Y_i \leq d_i \Delta_{i-1}$. From the stated bounds on the d_i's, we obtain the following more convenient bounds for Z_i, $i \geq 1$.

$$Z_i \leq (d_i)^2; \tag{3}$$

∎

The following lemma shows that the partial input produced by the processing in stages has few positions set to 0 or 1.

Lemma 3.3 *Let $b^0 = \{*\}^n$ and $b^1, \ldots, b^k$ be the partial inputs produced by the above stage by stage processing. Then b^k has at most $4P/m$ positions whose value is not $*$.*

Proof. We will show that the number of positions set to 0 or 1 in STAGE i is at most $P/m2^{i-2}$, which implies the bound in the lemma.

After the processing at stages $1, 2, \ldots, i-1$, by Lemma 3.2 a processor, p, after the read at level $i-1$ can be in at most Z_{i-1} states. Thus there can be at most Z_{i-1} edges between levels $i-1$ and i labelled p. Hence, the number of edges between levels $i-1$ and i is at most $Z_{i-1}P$, where P is the number of processors. Let H_i be the number of high degree vertices at level i. Then we have

$$H_i \leq \frac{Z_{i-1}P}{d_i}$$

The number of bits set in STAGE i is at most $N_{i-1}H_i \leq \frac{Z_{i-1}N_{i-1}P}{d_i}$.

Since $d_i \geq d_{i-1}^6 \geq Z_{i-1}N_{i-1}m2^{i-2}$ using (2) , (3) and $d_i \geq m2^{i-2}$, the number of bits set in STAGE i is at most $\frac{P}{m2^{i-2}}$. ∎

3.3 The final argument

Recall that $d_i = m^{6^{i-1}}$.

Theorem 3.1 *Let $c \geq 1$ be a constant, $\lambda \geq 0$ and $k = \lceil \frac{1}{4} \log(\frac{\log n}{\log(32c\lambda^2)}) \rceil$. Then, any algorithm that solves λ-Approximate Compaction with cn processors requires k steps.*

Proof. We prove the lower bound for $\lambda \geq 2$; notice that this implies a lower bounds for all smaller values of λ. For $\lambda \geq n^{\frac{1}{20}}/\sqrt{32c}$, the lower bound is trivial, so assume $2 \leq \lambda < n^{\frac{1}{20}}/\sqrt{32c}$.

Let A be an algorithm that claims to solve λ-Approximate Compaction with $P(n) = cn$ processors in less than k steps.

We choose m such that $\frac{4P(n)}{m} \leq \frac{n}{8\lambda^2}$; it suffices to choose $m = 32c\lambda^2$. Thus d_i is defined for $i \geq 1$. The choice of k in the theorem ensures that $m^{6^k} \leq \frac{n}{8\lambda^2}$.

For these choices of m and k we carry out the preprocessing described in the previous section. Let b be the partial input obtained; by Lemma 3.3 there are at most $\frac{4P(n)}{m} \leq \frac{n}{8\lambda^2}$ bits set in b.

In $X(b)$, there is some $x^* \in \{0,1\}^n$ such that x^* has exactly $\frac{n}{2\lambda} + 1$ 1's. For input x^*, some output cell among $\beta_{\frac{n}{2\lambda}+1}, \ldots, \beta_{\frac{n}{2}+\frac{n}{\lambda}}$ has a 1 written to it. For, if not, there must be a 1 written to β_i, for some $i > \frac{n}{2} + \frac{n}{\lambda}$. For the admissible values of λ, $\frac{n}{2} + \frac{n}{\lambda} > (\frac{n}{2\lambda} + 1)(1 + \lambda)$. Thus the padding factor is greater than λ, contradicting the claim of the algorithm.

Thus, in $G(b)$, there is an output cell β_j, for some j, $\frac{n}{2\lambda} + 1 \leq j \leq \frac{n}{2} + \frac{n}{\lambda}$, for which 1 is a possible content. We fix this cell to 1 using $FixCell(x^*, b, (\beta_j, k))$. By Lemma 3.1, the number of bits set by $FixCell$ is at most $N_k \leq d_k^2$. From the bounds on the values of the d_i's and the choices of m and k we have $d_k^2 \leq m^{6^k} \leq \frac{n}{8\lambda^2}$. Thus b' has at most $\frac{n}{4\lambda^2}$ bits set. Further, $\forall x \in X(b')$, output cell β_j has a 1 written in it.

Let b'' be the input obtained by setting all the $*$'s in b' to 0. Then b'' has at most $\frac{n}{4\lambda^2}$ 1's, and since $\frac{n}{2\lambda} + 1 > \frac{n}{4\lambda^2}(1 + \lambda)$, The 1 in β_j is outside the initial locations into which the 1's are to be compacted. Thus the padding factor is greater than λ and we have a contradiction. Thus A cannot solve λ-Approximate Compaction in k steps. ∎

By substituting a constant for λ in the expression for k in Theorem 3.1, we have

Corollary 3.1 *(Linear Approximate Compaction) For any constant λ, λ-Approximate Compaction requires $\Omega(\log\log n)$ time.*

Corollary 3.2 *For $\epsilon \to 0$, and $\lambda = n^\epsilon$, λ-Approximate Compaction cannot be done in constant time.*

Proof. When $\epsilon < 1$, for sufficiently large n, $32c\lambda^2 \leq \lambda^3$. Thus the expression in Theorem 3.1 for k reduces to $1/4\log(1/3\epsilon)$. As $\epsilon \to 0$, this expression goes to infinity. ∎

4 Remarks

We discuss the application of LAC to processor allocation. By approximately compacting them into $\mathcal{O}(k)$ initial cells, it is possible to allocate to each 1 a group of $\Omega(n/k)$ processors, such that the indices of each group form a contiguous sequence. It has been argued that the property of the processors forming a contiguous sequence is essential for processor allocation to be meaningful [9]. This is because if a processor allocated to a task does not know which other processors are allocated to the same task, it is difficult for it to co-operate effectively with the others. There is other evidence that supports this argument. Consider, for example, the ARBITRARY PRAM model. In this model, of the processors writing to a cell, any one may succeed. It has been shown that if a processor does not know the other processors in its group, then even simple tasks like finding the processor with the smallest index in the group are hard [18, 2].

On the other hand, on the PRIORITY model, the above task is easy. Functions like AND and OR can also be easily computed without knowing the other processors in the group. Thus a situation is conceivable in which, so long as a sufficient number of processors is allocated to each subtask, progress can be made, even if the processors do not know which other processors have been allocated to their task. The general problem of processor allocation may be formalized as follows. Given a 0-1 vector of length n as input, compute an output vector of p values, which has the following property: if j is the index of a bit in the input which has value 1, at least $\frac{p}{ck}$ output positions should have the value j, where k is the number of 1's in the input and c is a constant ≥ 1. We are to solve this problem with p processors. We think of the processors whose indices have value j as being allocated to the task with index j. Notice that this definition does not require the processors to form a contiguous sequence.

Our lower bound for LAC does not imply any bound for this problem. Its complexity is an interesting open question.

The technique of bounding the number of states that a processor can have, by setting input bits, is quite general. This technique was used in proving lower bounds for the chaining problem [5]. It has also been useful in proving lower bounds for a number of other problems in parallel computation [3]. We believe the methods are applicable to computational models other than PRAMs. This work is in progress.

5 Acknowledgements

We thank Jaikumar Radhakrishnan for his comments on an earlier draft and for pointing out the application to Corollary 3.2. We thank Torben Hagerup for many discussions and suggestions which greatly improved the paper.

References

[1] P. Beame and J. T. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, **36** (1989), pp. 643–670.

[2] S. Chaudhuri. Lower Bounds for Parallel Computation. *Ph.D Thesis*, Rutgers University, (1991).

[3] S. Chaudhuri. Sensitive Functions and Approximate Problems. *manuscript*, (1993).

[4] S. Chaudhuri, T. Hagerup and R. Raman. Approximate and Exact Deterministic Parallel Selection. In *Proc. of Math. Fdtns. of Comp. Sci.*, (1993), *to appear.*

[5] S. Chaudhuri and J. Radhakrishnan. The Complexity of Parallel Prefix Problems on Small Domains. In *Proc. 33rd Annual FOCS*, (1992), pp. 638-647.

[6] R. Cole and U. Vishkin Faster Optimal Parallel Prefix Sums and List Ranking *Information and Computation*, **81** (1989), pp. 334-352.

[7] J. Gil and L. Rudolph. Counting and Packing in Parallel. In *International Conference on Parallel Processing*, (1986), pp. 1000-1002.

[8] T. Hagerup. On a Compaction Theorem of Ragde. *Information Processing Letters*, **43** (1992), pp. 335-340.

[9] T. Hagerup. The Log-Star Revolution. In *Proc. 9th STACS, (1992)*, Springer LNCS, Vol. 577, pp. 259-278.

[10] T. Hagerup. Fast Deterministic Processor Allocation. In *Proc. 4th ACM-SIAM SODA* (1993), pp. 1-10.

[11] T. Hagerup. *personal communication.*

[12] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd IEEE FOCS* (1992), pp. 628–637.

[13] J. JáJá. An Introduction to Parallel Algorithms. Addison-Wesley, Reading, Mass., 1992.

[14] P.D. MacKenzie Load balancing requires $\Omega(\log^* n)$ expected time. In *Proc. 3th ACM-SIAM SODA* (1992), pp. 94-99.

[15] Y. Matias and U. Vishkin. Converting High Probability into Nearly-Constant Time - with Applications to Parallel Hashing. *In* Proc. 23rd Annual STOC, (1991), pp. 307-316.

[16] I. Newman, P. Ragde and A. Wigderson. Perfect Hashing, Graph Entropy and Circuit Complexity. In *Proc. Fifth Ann. Conf. on Structure in Complexity Theory*, (1990), pp. 91-99.

[17] P. Ragde. The Parallel Simplicity of Compaction and Chaining. *In* Proc. 17th ICALP (1990), Springer LNCS, Vol 443, pp. 744-751.

[18] P. Ragde. Processor-Time Tradeoffs in PRAM Simulations. *Journal of Computer and System Sciences*, (1992).

[19] L. Rudolph and W. Steiger. Subset Selection in Parallel. In *Proc. of the 1985 Int. Conference on Parallel Processing*, (1985), pp. 11-13.

[20] R. Sarnath. Lower bounds for padded sorting and approximate counting. TR 93-02, SUNY Buffalo, (1993).