Fast Load Balancing on a PRAM

by

Joseph Gil

Technical Report 91-14
June 1991

Department of Computer Science
University of British Columbia
Vancouver, B.C.
CANADA V6T 1Z2

Fast Load Balancing on a PRAM

Joseph Gil *†
The University of British Columbia
Email: yogi@cs.ubc.ca

January 1991

Abstract
We consider the following problem: n processors of a PRAM are given n independent tasks. Each task can be executed in constant time by a single processor. The distribution of tasks among the processors is unknown; each processor has information only about its set of tasks. The ***batch execution*** problem is to reschedule the tasks so that quickest execution of all the tasks will be achieved. Ignoring the overhead required for determining the redistribution pattern and the redistribution time itself, the tasks' execution can be done in $O(1)$ time. Thus the batch execution problem captures a basic cooperation obstacles of the PRAM model.

This paper solves the batch execution problem by using a novel object dispersal idea for crafting a *load balancing* algorithm (as well as several extensions): the load balancing algorithm moves tasks between the processors and outputs in an almost even distribution, i.e., when the algorithm completes its run, each processor has $O(1)$ tasks. The total run time is $O(\log \log n)$ with overwhelming probability. The algorithm and the techniques presented are expected to serve as useful building blocks in the design of other efficient parallel algorithms. Particularly, the load balancing algorithm can be employed as a general tool for achieving optimal speedup, and for eliminating scheduling difficulties from parallel algorithms.

*Part of this research was done while the author was at the Hebrew University of Jerusalem.

†Research supported in part by the Leibniz Center for Research in Computer Science, Jerusalem, Israel.

1 Introduction

In recent years considerable research has been devoted to CRCW-PRAM algorithms which run in less than $\log n / \log\log n$ time. Those algorithms are particularly interesting as they highlight and characterize problems which can be solved without counting (such problems are sometimes called "highly parallelizable" see [2]). One useful tool for evading the counting barrier is the usage of randomness to *approximate* quantities, as was done by Willard [19]: using a single concurrent-write memory cell, the number of participants in a game, n, can be estimated to within a constant factor after $O(\log\log n)$ rounds. A question that arises here is whether there are other "counting" problems which can benefit from randomization. This paper shows how to utilize randomization to give an approximate solution for the *batch execution* problem which was previously solved using the prefix sum algorithm:

THE BATCH EXECUTION PROBLEM

Before: m independent tasks are distributed in an unknown way among n processors of a PRAM, and each task can be executed in constant time by a single processor. Each processor has information only about its set of tasks.

After: All tasks are executed.

The batch execution problem occurs naturally in the design of parallel algorithms: Cole and Vishkin [6, 8, 7] pointed out the rescheduling difficulty in the general *accelerating cascades* algorithmic technique. Chlebus, Diks, Hagerup and Radzik [4, 5] encountered the same problem in the design of their sub logarithmic simulations between the CRCW-PRAM models. The lower bounds set by Gil, Meyer auf der Heide and Wigderson [14] show that efficient batch execution is the principal bottleneck in reducing the time of creating a linear size hash table for n keys from $O(\log\log n)$ to $O(\log^* n)$. Generally speaking, the batch execution problem occurs in converting algorithms which operate on an abstract model of computation (such as the parallel comparisons model) into PRAM algorithms.

By examining the problem we find that it has two major components: *work redistribution* and *task execution*. If the number of tasks is much larger than the number of processors, then the execution time dominates the redistribution time. If $m = n \log n$ then the execution time is at least $\log n$, and the standard parallel prefix sum algorithm [11] can do the work redistribution in $O(\log n)$ time. The problem becomes more challenging as m approaches n, if optimal speedup must be preserved. If $m = n \log n / \log\log n$ then the sophisticated $O(\log n / \log\log n)$ time optimal prefix sum algorithm of Cole and Vishkin [9] should be used for work redistribution.

Clearly, the most interesting case is $m = n$: in this case the actual task execution time can be constant, and the communication and cooperation difficulties manifest themselves in the run time of the algorithm. Let us therefore concentrate on the following *object balancing* problem

THE OBJECT BALANCING PROBLEM

Before: m objects are distributed in an unknown way among n processors of a PRAM. Each processor has information only about its set of objects.

After: The same objects are redistributed among the same processors. and there is a constant C such that each processor has at most Cm/n objects.

The above does not define exactly how objects are represented. Assume that the representation is such that processors can manipulate sets of objects in one time step, (a more precise statement of this assumption is given later). If the objects are tasks then the problem is referred to as the *load balancing* problem.

Our strategy for tackling the batch execution problem for the case $m = n$ is to first solve the load balancing problem and then let the processors continue executing their newly allocated tasks. We also show how the algorithm can be generalized to other values of m. The number of steps is $O(\log\log\min(n, m))$ with probability at least $1 - n^{-\epsilon}$ for some fixed $\epsilon > 0$ (this property renders the algorithm useful for embedding as a procedure in other algorithms).

The model of computation used is the ROBUST sub-model of the CRCW-PRAM introduced by Hagerup and Radzik [16], which can be thought of the weakest possible model of the CRCW. In this model if two or more processors try to write concurrently into the same cell, then the value actually written cannot be predicted. Thus, although concurrent writing is permitted, it cannot be used directly in the computation. Note that in the CREW model which forbids concurrent write, sub-logarithmic algorithms cannot be achieved [10].

The algorithm itself and a renaming procedure used in it are expected to serve as useful building blocks in the design of other efficient parallel algorithms. In particular the load balancing algorithm can be employed as a general tool for achieving optimal speedup: Gil and Matias in [13] show how the load balancing algorithm given in this paper can be used to achieve optimal speedup of parallel algorithms. The scheme presented there is rather general and can be used to achieve optimal speedup in other algorithms in which the processors$\times$time product is large, but the total number of *actual* operations satisfies the optimal speedup requirement. The number of processors can be reduced using the well-known Brent's theorem. Each processor simulates the assignments of several others in the original algorithm. Periodically, task reallocation is used to achieve a better balance between the number of simulated processors. This scheme is most useful in algorithms which have the property that if a processor becomes idle it is never activated again (a famous example of such algorithms is the list ranking algorithm due to Cole and Vishkin [8]). The load balancing algorithm may be used for balanced allocation of other resources (such as memory cells). A generic example is the allocation of processors to edges in sparse graphs algorithms, where the input is given by vertices.

The rest of this paper is outlined as follows: in Section 2 the main technique for object dispersal is described. Section 3 gives some necessary definitions. The results obtained are presented in Section 4, while Section 5 gives the details of our main load balancing algorithm.

2 The Dispersal Technique

The new dispersal idea which forms the basis for our load balancing algorithm for the case $m = n$ can be informally described as follows: suppose that there is a number u, such that no processor has more than u^2 tasks, then the following pseudo algorithm Disperse(u) redistributes the tasks so that no processor will have more than $2u$ tasks.

Algorithm Disperse(u)

A processor is said to be *loaded* if it has u tasks or more. (There are at most n/u loaded processors.) An auxiliary array of size n/u is allocated and then an injective mapping from the set of loaded processors to the array positions is constructed. Each loaded processor moves its task set to the auxiliary array position it is mapped to. The processors array is now partitioned into n/u *teams* where each team is associated with one auxiliary array position. The u members of the team take equal shares from the set of (at most u^2) tasks residing in this position.

Note that this sharing process adds at most u tasks to a processor, and since no processor had more than u tasks (otherwise it was loaded and its task set was to the auxiliary array), we can conclude that in the end of Disperse there will be no processor with more than $2u$ tasks.

In the initial unknown distribution, Disperse is applied by setting $u = \sqrt{n}$. Next, Disperse is applied iteratively. The upper bound on the maximal number of tasks after one application Disperse serves as an input parameter to the subsequent application. It can be shown that after $O(\log\log n)$ applications of Disperse all processors will have $O(1)$ tasks. Moreover, this double logarithmic behavior is preserved under very general perturbations in the definition of the Disperse algorithm.

Comment We say that a load balancing algorithm is exact if it operates with $m = n$ tasks and if it results in exactly one task for each of the processors. If $u = 2$, then an application of Disperse does not improve the distribution, and hence an exact load balancing cannot be achieved using Disperse. Exact load balancing is possible in $O(\log n/\log\log n)$ time using a prefix sum algorithm. The following lemma explains why only "approximate" load balancing is possible in time which is $o(\log n/\log\log n)$, by showing that an exact load balancing is at least as hard as counting.

LEMMA 1. *If there is an exact load balancing algorithm running in time $T(n)$ using a polynomial number of auxiliary processors, then there is another algorithm using a polynomial number of processors that will compute the number of* 1*'s in a binary array of size n.*

Proof We construct Alg_k, an algorithm which in $T(n) + O(1)$ time checks if the input array has exactly k 1's. Let k, $n/2 < k \leq n$ be fixed. Alg_k begins by creating a processor P_i for each position i, $1 \leq i \leq n$ in the input array. The processor P_i is assigned a single object if the input array had 1 in position i. Next, for all i, $n/2 < i \leq n$, if P_i was assigned an object it moves it to $P_{i-n/2}$. Algorithm $\mathsf{Alg}(k)$ is then employed for an exact load balancing among $P_1, \ldots, P_k$. Clearly $\mathsf{Alg}(k)$ will fail if the number of 1's in the input array was not k and such failure can be detected in constant time.

For k, $1 < k \leq n/2$, Alg_k, is implemented by flipping the bits of the input array and calling Alg_{n-k}. Algorithm $\mathsf{Alg}'(n)$ executes $\mathsf{Alg}_1, \ldots, \mathsf{Alg}_n$ in parallel and then determines in constant time which one of them succeeded. $\square$

The lower bounds of Beame and Hastad [1] thus preclude the existence of a very fast exact load balancing algorithm. It should be clear that the "approximate" balancing achieved by our algorithms is sufficient for almost all applications.

2.1 Realization of Disperse

A concrete implementation of **Disperse** must devise a fast way for creating the injective mapping between the loaded processors and the auxiliary array cells. In case there are almost n/u loaded processor, this mapping is closely related to counting and as such cannot be very fast. Thus it is necessary to increase the auxiliary array by (at least) a constant factor. The mapping itself is created by using a random selection process which is the core of indeterminacy in the load balancing algorithm. We will show that this mapping can be done in $O(\log\log n)$ time using a *renaming* algorithm. The renaming algorithm is of independent interest and it is expected to be applicable as a general parallel algorithms tool.

Consecutive applications of the **Disperse** algorithm require split and union operations on sets of tasks. Even if the original sets can be manipulated in constant time, then eventually processors will have to operate on a collection of set fragments. It can be shown that there are at most $2^{O(\log\log n)}$ such fragments in each collection, and by building a balanced binary tree on those task sets, manipulation of collections (intermediate task sets) requires $O(\log\log n)$ steps.

Thus we get that each invocation of the **Disperse** algorithm can take $O(\log\log n)$ time, which results in an $O((\log\log n)^2)$ load balancing algorithm. The goal of $O(\log\log n)$ time is reached based upon the observation that the renaming algorithm is very suitable for pipelining: most of the processors find a proper mapping after very few steps of the renaming algorithm. Therefore, after several steps of the renaming procedure of one instance of **Disperse**, the input condition of the following instance of **Disperse** is "almost" true, and this instance may start its execution. The task set manipulation overhead is circumvented by building a dispersal framework in which a processor assists others at most once throughout the algorithm.

3 Preliminaries

3.1 Task Distribution

Formally, a *task distribution* D is a set of m tasks and their allocation to the n processors. A processor P_i knows only D_i, the *set* of tasks allocated to it by D. We say that a task distribution D is *flat* if $\max_{1 \le i \le n} |D_i| = O(m/n)$. The task distributions D and D' are *equivalent* if they have the same task sets.

The input to a processor P_i consists of some identification for D_i, the ith input set, as well as m_i, its size. The load balancing algorithm never accesses the tasks themselves during its execution, and all necessary data structures are built on-line. Reference to tasks is done through task *bundles* of the form $\langle i, j_1, j_2 \rangle$, which stands for tasks j_1 through j_2 in the ith input set. Throughout the algorithm we let both j_1 and j_2 assume non-integer values, thus allowing reference to task fragments. The algorithm output is a set of at most two bundles for each processor. To ensure that no task is allocated to more than one processor in the output stage, we associate a task with a processor if that processor "owns" its lower fragment. This clearly does not increase the final load of a processor by more than one task.

The bundle notion implies that the exact representation of the set, and the way its identification is passed to a processor, is unimportant as long as the input sets are ordered and there is an access method to tasks numbered $j_1, \ldots, j_2$. We can conveniently think of each of the input sets as being represented by an array of tasks descriptions, where a set's identification is the memory address of the first array cell. This simple representation gives an $O(1)$ access time method to the jth task of a task set.[1]

3.2 Solid Algorithms

DEFINITION 1. *A probability is called* n-negligible *if it is smaller than $n^{-\epsilon}$ for some $\epsilon > 0$.*

We will also be talking about dominant probabilities (the complement of negligible probabilities) and about negligible and dominant events.

DEFINITION 2. *A probabilistic algorithm for a given problem is called* solid *if the sum of failure probabilities of all its steps is an n-negligible probability where n is a parameter describing the problem size.*

The union of a poly-logarithmic number of negligible events is also a negligible event. If the success of each step in the algorithm is dominant and the total number of steps is a poly-logarithmic then the whole algorithm is solid. A useful property of solid algorithms is that each step of a solid algorithm may safely assume that *all* previous steps succeeded.

3.3 Teams vs. Anonymous Sets

Two extremities of the knowledge state of members of processors sets are characterized by

[1]Not all the arrays of tasks are consecutive in memory or else the problem is trivial.

DEFINITION 3. *A set of processors Φ is* anonymous *if every processor knows whether or not it belongs to Φ, but no other information (such as cardinality, ordinal place etc.) is available to the set members.*

DEFINITION 4. *A* team *is a set of processors with consecutive indices, such that the starting and the ending index is known to all the set members.*

The regular structure of the team makes it possible to use it as a sub-PRAM. Assume that processors are divided into teams and that each team has a private memory of linear size in the cardinality of the team. This assumption does not add more than $O(n)$ memory to the whole machine. Mapping an anonymous set into a team of size N is done by solving

THE RENAMING PROBLEM

Before: An anonymous set Φ of processors, $|\Phi| \leq N$.

After: Each processor $P_i \in \Phi$ knows a value (the new name) $x_i \in \{1, \ldots, M\}$.
For $P_i, P_j \in \Phi$, $x_i \neq x_j$ if $i \neq j$.

Processors not in Φ do not participate in the computation.

4 Results

4.1 Renaming

The basic step of the renaming algorithm uses the following lemma, which will be used in the load balancing algorithm too.

LEMMA 2. *For any $N < M$ there is a way to execute a random name selection process, so that the number of processors that failed to find a new name is $< 2N^2/M$ with M-dominant probability.*

Proof Examine the following ranges of N for an appropriate fixed $\epsilon > 0$:

The high range $N > M^{0.5+\epsilon}$ The lemma follows from Chebeyshev's inequality and properties of random throws ([12] Lemma 2.9).

The low range $N < M^{0.5-\epsilon}$ In this case the expected number of colliding processors is $M^{-2\epsilon} < 1$. The *minimal* number of colliding processors which exceeds this is 2. The lemma in this range follows therefore from a simple application of Markov's inequality.

The intermediate range $M^{0.5+\epsilon} \leq N \leq M^{0.5+\epsilon}$ In this range, the throw must be carried out in *two* rounds. In the first round all N processors select a random name from a range of size $M/2$. In the second round only processors colliding in the first round participate, and they use the remaining $M/2$ names for random selection.

The expected number of colliding processors in the first round is at most $4M^{\epsilon}$. From Markov's inequality we infer that the actual number of colliding processors is $\leq M^{0.5-\epsilon}$

with M-dominant probability. Thus we can safely assume that we are in the *low range* of M in the second round, and that the final number of colliding processors is $\leq 2 < 2N/M^2$ with N-dominant probability.

□

THEOREM 1. *For a renaming problem with $M = O(N)$, there exists a solid algorithm* Rename, *which runs in $O(\log\log N)$ time on* ROBUST CRCW-PRAM.

Proof The algorithm behind this theorem uses an auxiliary array A of size $M = 16N$ to assign names for the processors. In iteration t ($t = 1, 2, \ldots$) all processors that did not yet reserve a private name try to do so by selecting a random place in a segment of size $16N/2^t$ of A, and then attempting to write their (old) name into it. The new name is assigned to a processor if it was the only writer to the cell.

Denote by $N(t)$ the number of active processors in the beginning of iteration t. The analysis is carried by setting a carefully chosen bound upper for $N(t)$ and showing that it holds with high probability. More specifically, let $\mathbf{N}(t) = 2^{-2^{t-1}-t+2}N$, then the event $N(t) \leq \mathbf{N}(t)$ for all $t \geq 1$ is N-dominant.

Initially $\mathbf{N}(1) = 2^{-2^0-1+2}N = 2^0 N = N$. Assume now that $N(t) \leq \mathbf{N}(t)$, by Lemma 2 we have

$$N(t+1) \;\leq\; 2\frac{2^{-2^t-2t+4}N^2}{16 2^{-t}N} = 2^{-2^t-(t+1)+2}N = \mathbf{N}(t+1) \;\; ,$$

which completes the induction step. Note that the number of iterations is $O(\log\log n)$, so even the last application of the basic renaming step uses $\Omega(N^\epsilon)$ memory for some $\epsilon > 0$. Hence, all iterations succeed with N-dominant probability. □

4.2 Load Balancing

As we have noted, if m is much larger than n, then the time needed to *execute* the tasks dominates the *reallocation* time, thus the most interesting case is $m = O(n)$. Our main algorithm deals with this case:

THEOREM 2. *There exists a solid algorithm which takes as input a distribution of n tasks among n processors, and after $O(\log\log n)$ time will output an equivalent flat distribution. The algorithm runs on* ROBUST CRCW-PRAM.

Proof The detailed algorithm description is given in the following section. □

The following lemma shows that even if execution time is disregarded, $m = O(n)$ captures the difficulty of reallocation.

LEMMA 3. *Suppose that there exists a load balancing algorithm* Alg *for $m = n$. Then* Alg *can be adapted for other values of m with the following overhead:*

$n = o(m)$: *$O(1)$ pre-processing time.*

$m = o(n)$: *$O(\log\log m)$ solid time pre-processing time.*

Proof

$n = o(m)$: Each processor divides the tasks it was initially allocated into blocks of m/n tasks, possibly leaving the last block incomplete. Blocks are then treated as "super tasks". There are no more than n incomplete blocks, so the number of super tasks can be bounded by $2n$. Algorithm **Alg** is then applied to reach a flat distribution of $2n$ super tasks over $2n$ virtual processors (each real processor plays the parts of two virtual ones). This output induces a flat distribution of ordinary tasks too.

$m = o(n)$: There are no more than m processors initially holding tasks, allowing us to use the solid renaming algorithm to compact in $O(\log\log m)$ time those processors into a segment of size $O(m)$. At this point **Alg** can be applied with parameter $n = m$.

□

As a consequence we get

THEOREM 3. *For any m, there exists a solid algorithm which takes as input a distribution of m tasks among n processors, and after $O(\log\log\min(n,m))$ time will output an equivalent flat distribution. The algorithm runs* ROBUST CRCW-PRAM.

Henceforth, $m = n$ is implicitly assumed unless otherwise stated.

5 The Main Load Balancing Algorithm

The full algorithm achieves a decrease in time to $O(\log\log n)$ by using simultaneous *pipelined* execution of a tailored version of the renaming algorithms. In this section we describe and analyze this pipelined execution.

5.1 Load Levels

We define a sequence of increasing *load boundaries* $L_1, \ldots, L_r$ by

$$L_{k+1} = L_k^{1.5}/2^k \qquad L_0 = 32 \ , \tag{1}$$

or in an explicit form

$$L_k = 2^{1.5^k+2k+4} \ . \tag{2}$$

A processor P_i is in *load level* k in iteration t if its current number of tasks $m_{i,t}$ satisfies $L_k \leq m_{i,t} < L_{k+1}$. If $m_{i,t} < L_1 = O(1)$ then P_i is considered to be unloaded, and the algorithm does not manipulate its tasks in any way. The highest load level is defined by $r = \min\{kL_{k+1} > n\}$, and clearly $r = O(\log\log n)$.

The (anonymous) set of all processors in load level k is denoted by Φ_k. Instance k of **Disperse**, denoted by **Disperse**$_k$, is devised for dealing with the load of the processors of Φ_k. The ideas behind the original **Disperse** algorithm are used in each renaming iteration of $\mathsf{Disperse}_k$ to distribute the tasks of each single processor at a certain load level to many processors at lower levels.

5.2 Pipelining Structure

In the first iteration of the algorithm only $\mathsf{Disperse}_r$ is active, the other instances of Disperse are activated gradually; $\mathsf{Disperse}_k$ begins working 3 iterations after $\mathsf{Disperse}_{k+1}$. The last instance to be activated is $\mathsf{Disperse}_1$. This activation delay lets the number of processors in high load level decreases significantly before the lower load levels are activated. This way, the "fallout" into a certain level from higher levels will be relatively small, allowing the renaming procedure of the low level to operate almost as if the "fallout" did not exist.

For each $\mathsf{Disperse}_k$ we have a *local* iteration counter, so while $\mathsf{Disperse}_k$ is in its *local* iteration t, $\mathsf{Disperse}_{k+1}$ is in its *local* iteration $t+3$ and the *global* iteration. counter is $3(r-k)+t$. It will be shown that for every load level k, the set Φ_k will be empty after $O(\log\log n)$ local iterations of $\mathsf{Disperse}_k$. Consequently, the number of global iterations needed to achieve a flat distribution is $O(\log\log n)$.

5.3 Virtual Processors

We view each processor as two virtual processors which correspond very much to the two different roles a processor may play in the Disperse algorithm. In any one step, each physical processor plays its two virtual parts in sequence.

1. Part of the processor's job is to dispose of its *original* task load. This part is played by the first virtual processor. If a processor initially has at least L_1 tasks, then the first virtual processor becomes active and it will remain so until it reserves an assisting team. The reserved team deactivates this virtual processor by removing *all* of its load.

2. Each processor is also a member of an assisting team. If a team is reserved to assist another processor, then all of its members gain a new bundle of tasks unrelated to their original load. The second virtual processor participates in the team and disposes of the *acquired* bundle of tasks. It is active from the time of the acquisition (provided that the acquired bundle has no less than L_1 tasks) until it finds an assisting team for itself. Only one instance of this virtual processor exists in any physical processor, because no assisting team can be reserved more than *once* throughout the algorithm.

The virtual processor concept simplifies the design and the understanding of the algorithm. The number of tasks of any virtual processor is constant throughout its "lifetime". The "death" of a virtual processor at a certain load level leads to the "birth" of a batch of new virtual processors at lower load levels. Henceforth, we refer to virtual processors as processors; no confusion will arise.

5.4 The Major Steps of the Algorithm

Allocate An auxiliary array A_k of size $2n/\sqrt{L_k}$ is used by $\mathsf{Disperse}_k$ for accumulating information about the anonymous set Φ_k. It is easy to verify that rapid growth of the sequence $\{L_k\}$ guarantees that the total size of all the auxiliary arrays is linear. The original size of Φ_k is by a simple counting argument no larger than n/L_k, so that by Theorem 1 an array of size $O(n/L_{k1})$ could have sufficed for renaming the set. The increase in array size to $2n/\sqrt{L_k}$ accounts for the fact that the set Φ_k dynamically changes; there is a constant flow of processors into Φ_k as a result of processor "disintegration" at higher load levels.

The array A_k is used in a manner similar to the usage of the auxiliary array in the renaming algorithm of Theorem 1: A_k is partitioned into segments, in iteration t ($t = 1, 2, \ldots$) a segment of size $n/(2^{t-1}\sqrt{L_k})$ is used by $\mathsf{Disperse}_k$.

Find An active processor P_i belongs to a fixed $\mathsf{Disperse}_k$. Using the global iteration counter, P_i determines the iteration number of $\mathsf{Disperse}_k$, and from that, the segment of A_k to be used. P_i chooses a random position in this segment, and tries to reserve it by writing its name into this position. If this reservation fails then P_i will be active in the next iteration.

Put Each P_i that managed to reserve a place in A_k moves its task bundle to that place. By the definition of load levels there is no position in A_k that contains more than L_k tasks.

Get The processors array is partitioned to $r+1$ *segments* of "assistants" that match the $r+1$ different load levels; segment k ($0 \le k \le r$) has $n/2^{k-1}$ processors which are all dedicated to sharing the load of processors in load level k.

Segment k is divided into *blocks*; the number of blocks equals the number of the positions of A_k. The processors in a block form an assisting team associated with a fixed position in A_k. In the **Get** step the team examines this array position, and if the task bundle in it is non-empty, the bundle is distributed among all the team members, each one taking an equal share.

Before proceeding to the time analysis, we need to ascertain two properties of the algorithm:

- The partitioning of the auxiliary array to segments ensures that no assisting team will assist more than once during the algorithm. Thus each virtual processor has exactly one task bundle to manage.

- The number of processors in an assisting team for load level k is
$$\frac{n/2^{k-1}}{2n/\sqrt{L_k}} = \frac{\sqrt{L_k}}{2^k} \quad .$$
A processor in such a team may become active with a number of tasks bounded by the maximal number of tasks of this load level divided by the size of the assisting team:

$$L_{k+1} \times \frac{2^k}{\sqrt{L_k}} = \frac{2^k L_k^{1.5}/2^k}{\sqrt{L_k}} = L_k \quad .$$

Thus an assisting processor will always enter a load level lower than the load level of the processor to which the assistance was given.

5.5 Time Analysis of the Algorithm

Let $N(t,k)$ be the size of Φ_k in the beginning of iteration t of algorithm $\mathsf{Disperse}_k$ (or in other words, iteration number $t+2(r-k)$ of the whole algorithm). A carefully chosen function $\mathbf{N}(t,k)$ is shown by induction to be an upper bound of $N(t,k)$; our bound will be

$$\mathbf{N}(t,k) = \frac{2^{1.5\cdot 2^t - t - 2}}{L_k^{2^{t-2}-0.5}} \times \frac{n}{L_k} \tag{3}$$

or in logarithmic form

$$\log \mathbf{N}(t,k) = 1.5\cdot 2^t - t - 2 - (2^{t-2} - 1.5)\log L_k + \log n \quad . \tag{4}$$

The induction base is given by:

LEMMA 4. *For a load level k, $N(1,k) \le \mathbf{N}(1,k)$.*

Proof Using (3) with $t = 1$ we get

$$\mathbf{N}(1,k) = \frac{2^{1.5\cdot 2^1 - 1 - 2}}{L_k^{2^{1-2}-0.5}} \frac{n}{L_k} = \frac{2^{3-3}}{L_k^{0.5-0.5}} \frac{n}{L_k} = \frac{n}{L_k} \quad ,$$

and, as was noticed before, the number of processors in load level k can never exceed n/L_k.

□

For the inductive step note that in the end of an iteration t, members of Φ_k can be classified into two types:

1. Processors that were in load level k before the iteration and did not leave it because they encountered a collision in their random selection.

2. Processors that joined load level k during the iteration as a result of assistance to processors in load level $k+1$ and higher.

Consequently $N(t+1,k)$ can be written as the sum of $N_c(t,k)$ (the number of processors of type 1) and $N_f(t,k)$ (the number of processors of type 2), and the induction step includes the two following parts:

LEMMA 5. *If $N(t,k) \le \mathbf{N}(t,k)$ then the event $N_c(t,k) \le \mathbf{N}(t+1,k)/2$ is dominant.*

Proof The proof uses Lemma 2 to estimate the number of processors that did not find an assisting team.

In the iteration, at most $\mathbf{N}(t,k)$ processors perform a random mapping into an array segment of size $n/(2^{t-1}\sqrt{L_k})$. Using Lemma 2 we can write

$$N_c(t,k) \le 2\mathbf{N}^2(t,k) \times \frac{2^{t-1}\sqrt{L_k}}{n} \quad .$$

Using the definition (3) of $\mathbf{N}(t,k)$ we have

$$\begin{aligned}
N_c(t,k) &\le \frac{2^{1.5\cdot 2^{t+1}-2t-3}}{L_k^{2^{t-1}-1}} \times \left(\frac{n}{L_k}\right)^2 \times \frac{2^{t-1}\sqrt{L_k}}{n} \\
&= \frac{1}{2} \times \frac{2^{1.5\cdot 2^{t+1}-(t+1)-2}}{L_k^{2^{(t+1)-2}-0.5}} \times \frac{n}{L_k} \\
&= \mathbf{N}(t+1,k)/2
\end{aligned}$$

□

LEMMA 6. *If $N(t+3i,k+i) \le \mathbf{N}(t+2i,k+i)$ holds for every $i \ge 1$ then $N_f(t,k) \le \mathbf{N}(t+1,k)/2$.*

Proof The proof is carried out by careful estimations on the "fallout" from higher load levels.

Let $N_f(t,k,i)$ denote the contribution of Φ_{k+i} $(i \ge 1)$ to Φ_k in iteration t of Φ_k, and let $\vartheta(t,k,i) = \log N_f(t,k,i) - \log \mathbf{N}(t+1,k)$. The proof shows that the contributions from higher load levels are bounded by a decreasing geometrical series, or more precisely that $\vartheta(t,k,i) < -(i+1)$ from which the lemma immediately follows.

Note that while load level k is in its iteration number t, load level $k+i$ is in iteration $t+3i$, thus $N_f(t,k,i)$ can be bounded by the product of $\mathbf{N}(t+3i,k+i)$ (the number of processors currently in load level $k+i$) by $\sqrt{L_{k+i}}/2^{k+i}$ (the size of the assisting team of load level $k+i$). Writing this bound in terms of $\vartheta(t,k,i)$ we get

$$\begin{aligned}
\vartheta(t,k,i) &\le \log \mathbf{N}(t+3i,k+i) + \log\left(\sqrt{L_{k+i}}/2^{k+i}\right) - \log \mathbf{N}(t+1,k) \\
&\le \log \mathbf{N}(t+3i,k+i) - \log \mathbf{N}(t+1,k) + 0.5 \log L_{k+i} - i \quad .
\end{aligned}$$

Using the definition (4) of $\mathbf{N}(t+3i,k+i)$ and $\mathbf{N}(t+1,k)$:

$$\begin{aligned}
&\vartheta(t,k,i) \\
&\quad \le 1.5\cdot 2^{t+3i} - t - 3i - 2 - (2^{t+3i-2} - 1.5)\log L_{k+i} + \log n \\
&\qquad - 1.5\cdot 2^{t+1} + t + 1 + 2 + (2^{t-1} - 1.5)\log L_k \\
&\qquad - \log n + 0.5 \log L_{k+i} - i \\
&\quad = 1.5\cdot 2^{t+3i} - 1.5\cdot 2^{t+1} + 1 - 4i - (2^{t+3i-2} - 2)\log L_{k+i} + (2^{t-1} - 1.5)\log L_k \\
&\quad < 1.5\cdot 2^{t+3i} - (i+1) - (2^{t+3i-2} - 2)\log L_{k+i} + (2^{t-1} - 1.5)\log L_k \quad .
\end{aligned}$$

Using the fact that $2^{t+3i-2} - 2 \ge 4(2^{t-1} - 1.5)$ for $t, i \ge 1$ we have

$$\begin{aligned}
\vartheta(t,k,i) &< 1.5\cdot 2^{t+3i} - (i+1) - (2^{t+3i-2} - 2) \times \log(L_{k+i} - \frac{1}{4}\log L_k) \\
&< 1.5\cdot 2^{t+3i} - (i+1) - (2^{t+3i}/4) \times (\log L_{k+i} - \frac{1}{4}\log L_k)
\end{aligned}$$

A simple check shows that $\log L_{k+i} - \frac{1}{4}\log L_k \ge 6$ and this completes the proof. □

THEOREM 4. *The event $N(t,k) \leq \mathbf{N}(t,k)$ for all $t = O(\log\log n)$ and $0 \leq r \leq r$ is dominant.*

Proof By simultaneous induction on t and k, applying Lemma 4 as the induction base and Lemmas 5 and 6 in the inductive step. $\square$

Acknowledgments I wish to thank my advisor Prof. Avi Wigderson for fruitful discussions and encouragement. Valuable comments made by Yossi Matias, Aviad Cohen and Larry Rudolph are gratefully acknowledged too.

References

[1] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[2] O. Berkman, D. Breslauer, Z. Galil, B. Scheiber, and U. Vishkin. Highly parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, 1989.

[3] O. Berkman and U. Vishkin. Recursive *-tree parallel data structure. In *focs89*, 1989.

[4] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *Mathematical Foundations of Computer Science*, 1988.

[5] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. In *Fundamentals of Computation Theory, Intl' Conf*, August 1989.

[6] R. Cole. Parallel merge sort. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 511–516, 1986.

[7] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 478–491, 1986.

[8] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[9] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[10] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[11] M. J. Fisher and R. E. Ladner. Parallel prefix computation. *Journal of The Association for Computing Machinery*, 27:831–838, 1980.

[12] J. Gil. *Lower Bounds and Algorithms for Hashing and Parallel Processing.* PhD thesis, The Hebrew University of Jerusalem, Givat Ram 91904, Jerusalem, Israel, November 90.

[13] J. Gil and Y. Matias. Fast hashing on PRAM. In *2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 91.

[14] J. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *Proc. of the 22st Ann. ACM Symp. on Theory of Computing*, pages 244–253, 1990.

[15] J. Gil and L. Rudolph. Counting and packing in parallel. In *Proc. 1986 International Conference on Parallel Processing*, pages 1000–1002, 1986.

[16] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *Proc. of the 1990 Symposium on Parallel Algorithms and Architectures*, 90.

[17] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 729–743, 1990. Also in TR-158/89, Eskenasy Inst. of Comp. Sci., Tel-Aviv Univ. Israel, Dec. 1989.

[18] P. L. Ragde. The parallel simplicity of compaction and chaining. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 744–751, 1990.

[19] D. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM J. Comput.*, 15:468–477, 1986.