To appear in IEEE Transactions on Computers

PRAM Processor Allocation: A Hidden Bottleneck in Sublogarithmic Algorithms

Allen Van Gelder*
Stanford University

June 7, 1987

Abstract

The problem of dynamic processor allocation in PRAMs is discussed, and differentiated from that of static allocation. The version of the PRAM we consider, also called the CREW model, is a parallel computer with global memory accessible in unit time that allows concurrent reads, but requires exclusive writes. Two dynamic processor allocation problems for P processors are distinguished. One, called *assignment from numbers*, has an $\Omega(\log P)$ lower bound, even when the number of tasks is $O(\sqrt{P})$. The second, called *assignment from leaders*, has faster solutions. A constant time solution for $O(\sqrt{P})$ tasks was known; we generalize it to $O(P^{1-1/k})$ tasks in time $O(k)$. Handling $O(P)$ tasks requires a different approach and we offer an algorithm that runs in time $O(\log \log P)$, which we conjecture is asymptotically optimal within a constant factor. We discuss the implications of these two versions of dynamic allocation on sublogarithmic merge algorithms proposed in the literature.

1 Introduction

The term PRAM, which stands for *parallel random access machine*, has been used in the literature to designate a variety of theoretical parallel computation models that permit processors to access global memory simultaneously and in unit time. Each processor executes its own instruction stream. Borodin and Hopcroft [BH85] describe a hierarchy of parallel computation models. We shall be concerned with the model that allows concurrent reads of the same global memory cell by many processors, but requires exclusive writes; all processors can write simultaneously, but each must write into a unique global memory location. This has lately been called the CREW model (*concurrent read, exclusive write*) to distinguish it from the more restricted EREW model (*exclusive read, exclusive write*),

*Supported by NSF grant IST-84-12791.

sometimes called the PRAC (*parallel random access computer*), which requires both exclusive reads and exclusive writes to global memory. The reader is referred to [BH85, Ull84] for additional particulars of these standard models.

Some PRAM algorithms require *dynamic processor allocation*. By this, we mean that the processors' task assignments cannot be computed ahead of time using just a few parameters of the problem instance, but instead depend upon the actual progress of the computation. In contrast, the task assignments for certain other algorithms, such as bitonic merge, FFT, etc. [Ull84], can be computed easily from the parameters of the problem, such as number of data elements; we call this *static processor allocation* because it does not depend on the data itself. In cases of dynamic allocation, the overhead of computing these task assignments has frequently been ignored, on the reasoning that the allocation-related computations are simple in relation to other parts of the algorithm, so can be absorbed into the "real work" with at most an increase in some constant factor. However, when the "real work" can be done very quickly, then dynamic processor allocation becomes a possible bottleneck.

1.1 Previous Work

Valiant [Val75] demonstrated that merging two lists of lengths $m \leq n$ could be done in time $O(\log \log m)$ stages, each stage requiring constant time, with $\sqrt{nm}$ processors in the *parallel comparison tree* model,[1] but did not address the issue of processor allocation. Then Borodin and Hopcroft [BH85] showed that dynamic processor allocation in the CREW model was achievable in constant time per stage for a merge algorithm based on Valiant's approach, thus $O(\log \log m)$ time for the whole algorithm. A close look at their allocation method shows that it depends on three crucial properties of the particular underlying algorithm:

1. The number of processors P is linear in $n + m$, the sum of the lengths of the lists. (In the parallel comparison tree model, $\sqrt{nm}$ processors are sufficient.)
2. The number of tasks (recursive merge subproblems) to be dynamically allocated for the next stage was $A = O(\sqrt{P})$.
3. The processors were to be allocated in contiguous groups whose *leaders* could be calculated in constant time. That is, the array

$$b_0 = 0, b_1, b_2, \ldots, b_{A-1}, b_A = P$$

specifies that processors b_j through $b_{j+1} - 1$ are to work together on task j; this array was computable in constant time. Only knowing the *numbers* of processors to be assigned to each task would have been insufficient; it would take time $O(\log A)$ to determine the leaders.

[1] The parallel comparison tree model is more powerful than the CREW; it allows 3^P branching based on any set of P key comparisons at each step.

Thus care must be exercised before assuming that the Borodin-Hopcroft allocation algorithm can be used on other problems.

1.2 Notation and Assumptions

We shall use P to denote the number of processors, and we assume that they are indexed $0, 1, \ldots, P-1$, and know their own indices. We let A stand for the number of tasks, or subproblems, among which the processors are to be allocated; the tasks are numbered $0, 1, \ldots, A-1$. Arrays have zero-based indices, in general. It is often convenient to use the values A and P at the ends of arrays as "sentinels", to simplify boundary conditions; keep in mind that they do not designate an actual task or processor.

For timing estimates, we assume that the running time of PRAM algorithms (at least the ones we consider here) is dominated by global memory references; a global memory read or write takes unit time. We assume that local operation times, including comparisons of integers and addresses, are neglible. However, in merge and sort routines, we assume that the (local) comparison of data keys takes time C; if a key is not already in the processor's local memory, then the time to fetch it must be added. Processors have bounded local memory. We also assume that a Pascal-type "record" that stores several items can be accessed in one step, and that a single array element may be such a record.

2 Two Versions of the Processor Allocation Problem

We describe two superficially similar processor allocation problems that have a marked difference in difficulty in the CREW model. These can also be looked at as memory allocation problems by considering the array a below to contain assignments of segments of a P-element array to A tasks.

Problem 2.1: Assign from Numbers

INPUT:

1. The number of processors P and the number of tasks A.
2. The array $n_0, \ldots, n_{A-1}$ with $n_j \geq 0$ that specifies the number of processors assigned to each task, and sums to P.

OUTPUT:

1. The array $a_0, a_1, \ldots, a_{P-1}, a_P = A$ that specifies each processor's task assignment; processor i is assigned to task a_i.

SPECIFICATIONS:

Each output array element a_i is to contain j if and only if

$$\sum_{k<j} n_k \leq i < \sum_{k<j+1} n_k$$

Recall that task A and processor P do not exist; those array positions simplify boundary conditions. □

Problem 2.2: Assign from Leaders

INPUT:

1. The number of processors P, and the number of tasks A.
2. The array $b_0 = 0, b_1, b_2, \ldots, b_{A-1}, b_A = P$ with $b_j \leq b_{j+1}$. Actually, each array element is a pair (b_j, b_j^+) such that $b_j^+ = b_{j+1}$.

OUTPUT:

1. The array $a_0, a_1, \ldots, a_{P-1}, a_P = A$ that specifies each processor's task assignment; processor i is assigned to task a_i.

SPECIFICATIONS:

The b array specifies that processors b_j thru $b_j^+ - 1$ are to work together on task j. (If $b_j = b_j^+$, then task j has no processors assigned.) Processor b_j is called the *leader* for task j.

Each output array element a_i is to contain j if and only if $b_j \leq i < b_j^+$. Note that task A and processor P do not exist; those array positions simplify boundary conditions. □

In both problems, we assume that the input array contains what we call *arbitrary values*. For our purposes, an arbitrary value is one that cannot reasonably be computed locally by some processor that does not already have it; the only practical way to get it is to read it from global memory.

Theorem 2.1: Problem 2.1, *Assign from Numbers*, requires time $\Omega(\log A)$ in the CREW model.

Proof: Cook and Dwork [CDR86] have shown that computing the logical OR of A boolean items with any number of processors in the CREW model has an $\Omega(\log A)$ lower bound. It is easy to reduce the A-way OR problem to an instance of Problem 2.1. ▮

We shall see in the next section that Problem 2.2, *Assign from Leaders*, can be done much faster. Thus, when considering dynamic processor allocation in sublogarithmic CREW algorithms, it is important to distinguish between these two versions of the problem.

An alternative approach to processor allocation is to dedicate a processor to a certain data element (or fixed number of elements), and as the element migrates from problem to subproblem to subsubproblem, the processor "follows it around". (C. Leiserson terms this the *active element* approach.) However, as a rule, in order to know its precise function within the subproblem, the processor will have to know "its" data element's exact position within the overall data structure for the subproblem. With array data structures, this very often reduces to an equivalent of one of the allocation problems described above.

3 CREW Processor Allocation Algorithms

First we describe a simple generalization of the Borodin-Hopcroft allocation algorithm for Problem 2.2, *Assign from Leaders*. This algorithm is parameterized by the integer $k \geq 2$, which we call the *degree* of the algorithm. Degree 2 corresponds to [BH85].

Algorithm 3.1: Assign from Leaders by Generalized BH(k)

INPUT:

1. The number of processors P, and the number of tasks A, where $A \leq \lfloor P^{\frac{1}{k}} \rfloor^{k-1}$.
2. The array $b_0 = 0, b_1, b_2, \ldots, b_{A-1}, b_A = P$ with $b_j \leq b_{j+1}$. Actually, each array element is a pair (b_j, b_j^+) such that $b_j^+ = b_{j+1}$. Processor b_j is called the *leader* for task j.

OUTPUT:

1. The array $a_0, a_1, \ldots, a_{P-1}, a_P = A$ that specifies each processor's task assignment; processor i is assigned to task a_i.

PROCEDURE:

1. Each processor does the following:

   (a) Read A and P.

   (b) Compute $Q = \lfloor P^{\frac{1}{k}} \rfloor \geq A^{\frac{1}{k-1}}$.

   Note that $AQ \leq P$. Time = 1.

2. Form A groups of Q processors each. All processors in the j-th group compute $b_j^+ - b_j$. If this value is less than or equal to Q, then they write j in locations $a(b_j)$ through $a(b_j^+ - 1)$.[2] In this case, we call j a *short* task. Time = 2.
3. Form A groups of Q processors each. For $1 \leq i \leq A$, the i-th group will determine the "fencepost" assignment $a(iQ)$ in $k-1$ steps. The method is a $(Q+1)$-ary search [Kru83] on the b array to find the j such that $b_j \leq iQ < b_j^+$. Time = $4k - 6$.
4. For each processor i, let i_L be the index of the "fencepost" of step 3 at or below i; i.e., $i_L = Q\lfloor i/Q \rfloor$. Each processor i now checks whether j_L or j_U is its correct assignment, where $j_L = a(i_L)$ and $j_U = a(i_L + Q)$.

   - If $i_L > 0$ and $b(j_L) \leq i < b^+(j_L)$, then it sets $a_i = j_L$.
   - If $i_L < A$ and $b(j_U) \leq i < b^+(j_U)$, then it sets $a_i = j_U$.
   - If both j_L and j_U fail, then, as shown in [BH85], processor i has been assigned to a short task in step 2, and so reads its assignment from a_i.

   Time = 5.

5. Each processor i reads its new task boundaries, $b(a_i)$ and $b^+(a_i)$. Time = 1.

[2] To avoid double subscripts we denote array elements by $a(i)$ as well as a_i.

The total time required, which includes time for processors to get their new assignments and task boundaries into local memory, is $4k + 3$. □

As the number of tasks grows in relation to the number of processors, we see that the performance of the above algorithm degrades, as illustrated in the following table:[3]

A	$4k$
$P/\lg P$	$4\lg P/\lg\lg P$
$P/2^h$	$\frac{4}{h}\lg P$
P	∞

One way to handle values of A very close to P more efficiently is to balance the times of steps 2 and 3 in Algorithm 3.1 by defining a short task to be one assigned no more than wQ processors, for an appropriate positive integer w. Then in step 3 it suffices to find the assignment of every wQ-th element, using groups of wQ processors to speed up the search.

Algorithm 3.2: Assign from Leaders by Optimized BH

INPUT:

1. The number of processors P, and the number of tasks A, where $A \leq P$.
2. The array $b_0 = 0, b_1, b_2, \ldots, b_{A-1}, b_A = P$ with $b_j \leq b_{j+1}$. Actually, each array element is a pair (b_j, b_j^+) such that $b_j^+ = b_{j+1}$. Processor b_j is called the *leader* for task j.

OUTPUT:

1. The array $a_0, a_1, \ldots, a_{P-1}, a_P = A$ that specifies each processor's task assignment; processor i is assigned to task a_i.

PROCEDURE:

1. Each processor does the following:
   (a) Read A and P.
   (b) Compute $Q = \lfloor \frac{P}{A} \rfloor$.
   (c) If $\lg Q \geq 2\sqrt{\lg P}$, then set $w = 1$, reducing to Algorithm 3.1; otherwise, solve
   $$w = \frac{4\lg A}{(\lg Q + \lg\lg A)^2}$$
   approximately for an integer w.

   Time = 1.
2. Form A groups of Q processors each. All processors in the j-th group compute $b_j^+ - b_j$. If this value is less than or equal to wQ, then they write j in locations $a(b_j)$ through $a(b_j^+ - 1)$. In this case, we call j a *short* task. Time = $1 + w$.

[3] We use "lg" for base 2 log, and use "log" when the base is immaterial.

3. Form A/w groups of wQ processors each. For $1 \leq i \leq A/w$, the i-th group will determine the "fencepost" assignment $a(iwQ)$ in $k-1$ steps, where $k = \lg A/(\lg Q + \lg w)$. The method is a $(wQ+1)$-ary search on the b array to find the j such that $b_j \leq iwQ < b_j^+$. Time $= 4k - 6$.
4. For each processor i, let i_L be the index of the "fencepost" of step 3 at or below i; i.e., $i_L = wQ\lfloor i/wQ \rfloor$. Each processor i now checks whether j_L or j_U is its correct assignment, where $j_L = a(i_L)$ and $j_U = a(i_L + wQ)$.
   - If $i_L > 0$ and $b(j_L) \leq i < b^+(j_L)$, then it sets $a_i = j_L$.
   - If $i_L < A$ and $b(j_U) \leq i < b^+(j_U)$, then it sets $a_i = j_U$.
   - If both j_L and j_U fail, then, processor i has been assigned to a short task in step 2, and so reads its assignment from a_i.

   Time $= 5$.
5. Each processor i reads its new task boundaries, $b(a_i)$ and $b^+(a_i)$. Time $= 1$.

The total time required, which includes time for processors to get their new assignments and task boundaries into local memory, is $4k + w + 2$. In the cases where $w > 1$, at best $\lg Q$ is near $2\sqrt{\lg P}$ and we get a time of about $2\sqrt{\lg P}$. However, as Q gets significantly smaller (in relation to P) the time goes up to approximately $4 \lg A/(\lg Q + \lg \lg A)$. Note that w does not contribute to the leading term. See Fig. 1 for some sample values. □

As the number of tasks grows in relation to the number of processors, Algorithm 3.2 performs somewhat better than Algorithm 3.1, as illustrated in the table in Fig. 1.

We shall present a different allocation algorithm with better asymptotic performance in this range of A *vs.* P. But first, we describe a useful operation on sorted arrays.

Let x and y be sorted arrays, of lengths m and n, respectively. The operation *cross-rank*(x, y) consists of computing two new arrays, r and s, such that

- r_i is the rank of x_i in y, i.e., is the number of elements in y that are less than x_i.
- s_j is the rank of y_j in x.
- We assume that elements from different arrays never compare equal; if there is no other way to break a tie, the element from the first operand (x) is considered to be less.

Cross-rank is a frequently used building block in CREW algorithms for merging and sorting, since it can be done in time $O(\log \log(n+m))$ with $O(n+m)$ processors [Val75, BH85]. After x and y are cross-ranked, then their merger into a new array z is accomplished in constant time with $O(n+m)$ processors by moving x_i to $z(i + r_i)$ and moving y_j to $z(j + s_j)$. Less obviously, Preparata [Pre78] showed that k arrays of combined length n could be merged with kn processors by cross-ranking all $\binom{k}{2}$ *pairs* of arrays, then summing their individual ranks to obtain the merged ranks. With the choice $k = \lg n$, this leads to an $O(\log n)$ sort using $O(n \log n)$ processors. We shall show how cross-ranking can be

used for processor allocation in order to get good performance when the number of tasks is large in relation to the number of processors.

Algorithm 3.3: Assign from Leaders by Cross-Ranking

INPUT:

1. The number of processors P, and the number of tasks A, where $A \leq P$.
2. The array $b_0 = 0, b_1, b_2, \ldots, b_{A-1}, b_A = P$ with $b_j \leq b_{j+1}$. Actually, each array element is a pair (b_j, b_j^+) such that $b_j^+ = b_{j+1}$. Processor b_j is called the *leader* for task j.

OUTPUT:

1. The array $a_0, a_1, \ldots, a_{P-1}, a_P = A$ that specifies each processor's task assignment; processor i is assigned to task a_i.

PROCEDURE:

1. Initialize a work array $p_i := i$ for $0 \leq i \leq P$.
2. Perform cross-rank(b^+, p), giving output arrays t and a. (The work array t may be discarded; however, t_j contains the number of the next *nonnull* task above j, which may be useful in some problems.) For processor allocations required within the cross-rank procedure, use Algorithm 3.1. The discussion of Fig. 1 below shows why this algorithm should not be used recursively.

The time required is essentially 1 plus the time required for cross-ranking, which is analyzed in Section 4. We used $C = 0$ (recall that C is the time for local comparison of data keys), since the keys being compared were integers. Time $= 20 \lg \lg A + 8$. □

A CREW lower bound of $\Omega(\log \log n)$ to merge two n-element arrays was shown in [BH85]. Algorithm 3.3 solves Problem 2.2, *Assign from Leaders*, by treating it as a special case of merging, where one array is just consecutive integers. We do not see any way to take advantage of this special case, and conjecture that its lower bound is also $\Omega(\log \log n)$. If this conjecture is true, it follows that Algorithm 3.3 is asymptotically optimal within a constant factor when $A = O(P)$.

We see in Fig. 1 that Algorithm 3.3 asymptotically is faster than Algorithm 3.2 in all cases where the latter uses a value of w greater than 1. However, this is a theoretical victory only, as the cross-over point is somewhere in the neighborhood of $P = 2^{360}$.

4 A Closer Look at Kruskal's Merge

Kruskal [Kru83] gives a family of merge algorithms with parameter k, which we shall call the *degree*. The algorithms cross-rank two arrays of lengths m and n. We assume $m \leq n$. In the parallel comparison tree model he shows that the number of processors required is the sublinear quantity $\left(\frac{m}{n}\right)^{\frac{k-1}{k}} n$. However, his remark that [BH85] shows how to allocate processors for CREW versions of these algorithms is an overstatement, in view of the

Tasks (A)	Optimized BH ($4k + w + 2$)	Cross-rank ($20 \lg\lg A$)
$P^{1/2}$	11	$20 \lg\lg P$
$P/P^{1/k}$	$4k + 3$	$20 \lg\lg P$
$P/(2^{2\sqrt{\lg P}})$	$2\sqrt{\lg P}$	$20 \lg\lg P$
$P/\lg P$	$4 \lg P/\lg\lg P$	$20 \lg\lg P$
$P/2^h$	$4 \lg P/\lg\lg P$	$20 \lg\lg P$
P	$4 \lg P/\lg\lg P$	$20 \lg\lg P$

Figure 1: Asymptotic performance of allocation algorithms for the *Assign from Leaders* problem, where $k = \lg A/(\lg Q + \lg w)$ and h is a constant.

limitations mentioned in Section 1.1. However, with a linear number of processors, say $P = n + m$, Algorithm 3.1 performs the allocation in time that is linear in the degree, k, but independent of problem size.

It is interesting to re-analyze the performance of Kruskal's family of merge algorithms as a function of k, using global memory references as the measure, and accounting for processor allocation. Briefly, the algorithm of degree k runs in $\lg\lg m/\lg k$ "nonfinal" stages, plus a "final" stage. The "real work" of each nonfinal stage consists of k Q-ary searches to determine the exact ranks of equally spaced "samples" from the shorter array in the longer array. This allows one merge problem to be partitioned into A smaller merges in the next stage, where $A = P^{\frac{k-1}{k}}$. We estimate the time for the "real work" to be $(5 + C)k + 1$ for nonfinal stages. The final stage requires $7 + C$. However, the overhead time for processor allocation adds $4k + 3$ to the time for each nonfinal stage. Thus nearly half the time is spent in processor allocation!

By varying k we can trade off time per stage against the number of stages. In order to find the optimum choice of k, we minimize

$$T(m, k) = ((9 + C)k + 4)\frac{\lg\lg m}{\lg k} + 7 + C$$

with respect to k. Assuming $C \leq 1$, the value of T is about equally small when k is 3 or 4; these choices are about 10 percent better than $k = 2$. For computational ease, we use $k = 4$ as representative, giving the following expression for time to cross-rank two arrays of lengths $m \leq n$ with $n + m$ processors:

$$T(m) = (20 + 2C)\lg\lg m + 7 + C$$

We note in passing that the bitonic merge, or even-odd merge, upon which Batcher's sort is based [Bat68, Ull84], is easily implemented in time $(2 + C)\lg(n + m)$. Note the

low multiplicative constant of $(2 + C)$ *vs.* $(20 + 2C)$ for the sublogarithmic merge. This results from the "static" nature of the processor allocation for the bitonic merge: no global memory accesses are needed for processors to determine their tasks. If $C \leq 2$ and both arrays are about the same size, then the bitonic merge outperforms the sublogarithmic versions up to about $n = 2^{30}$. This conclusion would not be reached by only looking at the number of comparisons.

5 Conclusion

We have shown that processor allocation may play a significant role in sublogarithmic CREW algorithms. In addition, it appears that counting both comparisons and global memory accesses gives a significantly different picture of merging algorithms from that given by just counting comparisons. We leave as open questions whether $\Omega(\log \log P)$ is the lower bound for *Assign from Leaders* with $A = P$, and whether the constant factor can be improved.

We thank the referees for their careful reading of the paper.

References

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 307–314, 1968.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, 1985.

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, 1986.

[Kru83] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. on Computers*, C-32(10):942–946, 1983.

[Pre78] F. P. Preparata. New parallel-sorting schemes. *IEEE Trans. on Computers*, C-27(7):669–673, 1978.

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[Val75] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Computing*, 4(3):348–355, 1975.