Foundational Algorithms for Computational Distributed Robot Swarms

David Eppstein Michael T. Goodrich Nodari Sitchinava
University of California, Irvine
{goodrich, eppstein, nodari}(at)ics.uci.edu

Abstract

In this paper, we study discrete swarm algorithms, where mobile robots (or "mobots") move around interacting in an environment to solve computational problems. This work extends recent work on swarm algorithms in the distributed computing, artificial intelligence, and robotics literatures in that we assume that mobots are ***computational***, that is, they have additional memory and hardware so as to enable computations that are somewhat more sophisticated than finite state automata. We describe efficient swarm algorithms for a number of foundational problems, which are essential prerequisites for coordinated movement, data gathering, and data processing computations, including rectangle ranking, prefix sums and products, permutation routing, and sorting. Each of our algorithms is optimal in at least one complexity measure.

[Regular submission to SPAA 2008.]

1 Introduction

Motivated by recent developments with mobile sensors and nanobots, this paper is directed at the study of distributed algorithms for computing devices that move intelligently. Such collections of mobile robots (which are sometimes called ***mobots*** [7]) suggest an intriguing paradigm, where computing devices move around an environment to perform a computation, all without centralized control. For example, there might be a decentralized action that the mobots need to perform, such as a coordinated search for survivors of an avalanche or pieces of a crashed airplane. Additional examples of such environmentally-induced computations could include the detection and monitoring of hazardous spills, physical resource allocation, and military applications. We envision a computational paradigm where a swarm can perform all aspects of a computation, including setup, distributed search and data input, collective data processing, and post-processing action, all without centralized control.

The PMR Model. To facilitate the design and analysis of computational distributed swarm algorithms, let us define a parallel computational model that captures the essential features of how mobots can be used to collectively solve computational problems. Given some precision distance, based on robot size and sensing capabilities, this model discretizes the environment so as to provide a simple notion of location. Specifically, define a model, which we will refer to as the ***Parallel Mobile Robot*** (PMR) model, as follows:

- *Each mobot has storage capacity that is at least as large as* $\log m$*, where* m *is the number of mobots.* This allows mobots to at least be able to count each other, for example.
- *The mobot collection is coarsely synchronized.* The coarse-grained clock can be implemented using reasonably-synchronized internal clocks or an external wireless clock.
- *The environment is two-dimensional, with coordinates specified by integer* x*- and* y*-coordinates, each mobot knows its location in the environment, and each grid location can hold at most one mobot.*
- *In any step, a mobot may stay in its cell or it may move to an adjacent cell, but only if such a move is collision-avoiding.* If a mobot attempts to move into an adjacent grid cell that is occupied by another mobot, then it fails and stays in its cell. Likewise, if two mobots attempt to move into the same cell, then they both fail.
- *A mobot can communicate a single message to an adjacent mobot in one step.* There are a number of ways such communication can be implemented, including the use of infrared transmitters/receivers, RF antennas, or even the collision-avoidance rule[1].

Variations. With no extensions to the above rules, we define the ***unit-cost*** PMR model. We also can consider an extension of this model, however, to a variant we call the ***long-haul*** PMR model, which is motivated by mobile robots that can move quickly in a straight line:

- *A mobot can move horizontally or vertically from a point* p *to a point* q *in a single step.* This movement in the ***long-haul*** variant is allowed, however, only if no such paths cross. We allow such paths to overlap, but they may not share common endpoints.

Note that this rule only allows mobots on long-haul paths to follow one another. Some algorithms assume that a single vertical or horizontal long-haul track is implemented physically as a single lane. This assumption implies that long-haul paths cannot be completely contained inside other long-haul paths, for a long-haul path completely inside another means that a mobot could stop short in front of another, which would cause a crash. Other algorithms instead assume that long-haul tracks consist of a travel lane and a parking lane, in which case we can allow long-haul paths to be completely contained in other long-haul paths, since a mobot that has reached its destination can simply pull into the "parking spot" at that point (which must be open, since long-haul paths cannot share endpoints). Algorithms that do not use this mobot parking capability are called ***parking-free***.

[1] The collision-avoidance rule allows a pair of mobots separated by a single grid cell to "dance" so as to communicate bits, e.g., by having an attempt to move to an adjacent space be a '1' and the lack of such an attempt be a '0.'

Analyzing PMR Algorithms. In traditional parallel models, algorithms are typically analyzed in terms of running time, memory space, and processor count. For a PMR algorithm, specified by (small) programs stored in the mobots, there are various ways of measuring efficiency:

- *The number of mobots*. Typically, we assume that the number of mobots is equal to n, the input size.
- *The number of steps required by the algorithm*. We refer to this bound as the "running time." Note that the number of steps also provides a bound on communication complexity, which, in this paper, we focus on the case when mobots pass 1-bit messages in a single step (i.e., a bit-communication analysis).
- *The maximum distance traveled by any mobot*. This measure corresponds to the amount of power needed by each mobot to perform a given computation.

Previous Related Work. In the artificial intelligence (AI) and robotics literatures, there has been considerable work done on algorithms for collections of mobile robots (e.g., see [6, 21]). The main focus of this previous work has been on the study of emergent behavior of a collective from simple rules defined for the mobots. In fact, this motivation is the reason why such collectives are usually called ***swarms***. Interestingly, with only a handful of simple rules for the mobots, simulations of swarm algorithms show a remarkable ability to mimic the behavior of groups of ants, bees, fish, or birds [2, 15, 21, 22, 23, 24, 25, 26, 28]. Moreover, such algorithms have useful applications in film making, as they can be used to create lifelike moving images of large groups. For example, swarm algorithms were used to create some of the scenes in the "Lord of the Rings" movie trilogy (see `http://www.massivesoftware.com/`).

Although the subject of emergent collective behavior from simple rules is interesting and even profitable, our interest is complementary to existing work on swarm algorithms. That is, we are not focused in this project on getting mobots to act like ants, bees, fish, or birds, but are instead interested in models and algorithms for ***computational*** swarms, which describe in a step-by-step fashion how a collection of mobots can work together to solve specific computational problems. Interestingly, even at the model level, our approach contrasts with the prior work on swarm algorithms, for, in the PMR model, we allow for a limited internal memory for each mobot, which can be used to do computations internal to that mobot. Existing swarm models, in contrast, assume only a constant number of memory bits and restrict each mobot's computational strength to that of a finite state automaton (FSA) [7]. This limitation seems artificially restrictive in a world where a digital camera can store gigabytes on a removable card the size of a dime. Fortunately, some researchers in the robotics community are also recognizing this fact and are allowing swarming robots to have reasonably sized memories (e.g., see [3]). Even so, since cellular automata can simulate Turing machines [20], we note that a sufficient number of FSA-based mobots can simulate a Turing machine (basically by building it out of a large number of mobots). Thus, providing for limited mobot memory allows for discrete swarm algorithms to be more efficient and makes their analysis more realistic, but it doesn't extend the class of computable functions.

There has also been some prior work on ***distributed robot swarm*** computations, for robot dispersion, separation, search, and organization tasks. Hsiang *et al.* [12] study methods for dispersing mobile robots in an unknown environment, using a model that is similar to the PMR model but nevertheless restricts robot memory to constant-size and the control mechanism to an FSA. Wang and Zhang [30] study a simple separation task, where all the mobots are colored red or blue and they wish to separate into two clusters of identically-colored mobots. Arkin *et al.* [1] study the "freeze-tag problem," which involves "waking up" a collection of mobile robots to perform a collective task. They address the optimization problem of minimizing the running time of the algorithm, showing it is NP-hard, but approximation algorithms exist. Beslon *et al.* [5] study mobile robot searching strategies, where mobots are looking for desired locations while avoiding obstacles. Cieliebak *al.* [8] study the problem of bringing a collection of mobile robots to a common point. There is also previous work that solves mobot gather and dispersal problems by viewing mobots as particles with attractive and repulsive forces acting upon them (e.g., see [4, 9, 10, 19]). In spite

of this previous work on distributed swarm robots, some of which contains elements of the PMR model described above, we are not aware of previous work that studies computational swarm models with the features we consider here, including the modeling of mobots to have reasonable memory sizes and coarse-grain awareness of time, all of which are easily realizable with modern hardware.

There are also some observations that we can draw from existing work on parallel algorithms. For example, we can assign n mobots to grid locations in a $\sqrt{n} \times \sqrt{n}$ mesh and have them simulate a mesh-computer algorithm [17]. Once the mobots configure themselves into a square mesh, they can then communicate with their neighbors to simulate the mesh algorithm. Thus, any mesh algorithm can become a PMR algorithm with the number of steps in such a simulation being $\Theta(\max\{\sqrt{n}, T(n)\})$, where $T(n)$ is the (bit-complexity) running time of the mesh algorithm they are simulating.

Similarly, n mobots can simulate a uniform circuit, by physically traversing a "drawing" of it. This simulation is only possible for circuits that can easily be described in $o(n)$ space. In addition, a collection of mobots can easily simulate a set of cellular automata, which are grids of non-moving cells with finite state that can change state in reaction to the states of neighboring cells [14, 27, 31].

Our Results. In this paper, we give discrete swarm algorithms for the following problems:

- *Rectangle ranking*: sort mobots by location, so each mobot learns its lexicographic rank according to (starting) x- and y-coordinates. This subproblem is useful, for example, in solving mobot dispersion, compression, and coordinated search problems.
- *Prefix products*: compute the value of an associative function for each prefix of lexicographic ranks. This subproblem is useful, for example, in aggregating sensor data and in weighted ranking problems.
- *Coordinated motion*: route a collection of mobots so that they physically move to alternative positions specified by a permutation (that is, such that no two mobots have the same destination address).
- *Parallel sorting*: physically sort mobots by (stored) input values.

We believe these problems are foundational, in that arise naturally in the solutions to other swarm algorithms, which would naturally come in future work. Our algorithms for these problems are all optimal, in terms of their running times or the maximum distance traveled per mobile robot. In addition, our algorithms illustrate the additional power that can be achieved simply by giving mobile robots a small amount of memory (whose size depends on the number of mobots). For example, several of our algorithms make use of a technique where we have a group of mobots perform a computation that is a ***rehearsal*** for a future computation. Even with limited memory, the mobots can remember their previous rehearsal so that they can perform the future actual computation without conflicts.

2 Rectangle Ranking and Parallel Prefix Computations

Rectangle ranking and parallel prefix computations solve basic problems for swarms of mobile robots. For example, they form foundational subproblems in the problem of evenly distributing mobots in an environment or for performing a coordinated search for an object of value. In this section, we present several efficient algorithms in the PMR model for these problems.

The Rectangle Ranking Problem. In the ***rectangle ranking*** problem, we are given a configuration of n mobots (although they may not know the value of n) in an $N \times M$ grid, and we are interested in numbering the mobots from 1 to n based on a lexicographical ordering of their coordinates in this initial configuration.

The Prefix Products Problem. In the ***prefix products*** problem, we are given a configuration of n mobots in an $N \times M$ grid, and we are interested in computing, for each mobot x_i, the product $x_1 \otimes x_2 \otimes \cdots \otimes x_i$, where $\otimes$ is any associative operator and the subscripts are based on a numbering of the mobots from 1 to n in a lexicographical ordering of their coordinates in this initial configuration.

2.1 Solving the Prefix Products Problem in the PMR Model

We describe in this subsection a solution to the prefix products problem in the long-haul model that runs in $O((\log N + \log M)w)$ time with maximum distance $O(N + M)$, where w is the word size needed to store values of the $\otimes$ operator. This result immediately implies a solution in the unit-cost model that runs in $O(N + M)$ time for reasonable values of w.

Expand the $N \times M$ grid to a $2N \times 2M$ grid, by having each mobot move to the x-coordinate double its own and then the y-coordinate double its own (so there is an empty row and column next to every mobot). Imagine that we define a complete binary tree on the cells of each occupied row of the grid and then a complete binary tree on these rows. Let T denote this tree (even though we don't explicitly store T anywhere). Each node v of T is associated with a subgrid of our environment, which is the union of all the cells that are descendents of v plus the empty cells above and to the right of these cells. In addition, let us associate with each such v a center point $p(v)$ in an empty row of v's subgrid, and note that each such center point will have a grid cell to the left of $p(v)$ and a grid cell to the right of $p(v)$. Based on its initial position, each mobot is assigned to a leaf of the tree T. At a high level, the algorithm for prefix products consists of the following two phases, which mimic the parallel prefix algorithm of Ladner and Fischer [16]:

1. In the first phase, which we call the ***up-sweep*** phase, the collective computes, for each internal node v of T, the product of values stored in mobots that are in cells associated with descendents of v.
2. In the next phase, which we call the ***down-tree*** phase, the collective computes, for each node v of T, the product of values stored at mobots in the preorder leaf predecessors of v.

The final values associated with the leaves of T are the solutions to the prefix products problem.

The Up-sweep Phase. Suppose inductively we have a mobot associated with each node v of T on level i that has at least one mobot in a descendent cell. Each such mobot x moves to the (appropriate left/right side of the) center point of the subgrid of size $N(v) \times M(v)$ associated with v's parent in T. This movement can be done in $O(1)$ time in the long-haul model, using the empty rows and columns like "streets," with the distance traveled being $O(N(v))$ during the horizontal combination rounds and $O(M(v))$ during the vertical combination rounds. If x meets another mobot there, then the two mobots are designated to exchange information, i.e., x shares its product value associated with v and receives the product value from the mobot for v's sibling in T. This information exchange can be done using the bit-communication rule to run in $O(w)$ time. If there is no other mobot waiting at the meeting point, then v's sibling has no mobots associated with its descendents and its product value is taken to be 1. At the same time, one of the communicating mobots is chosen as a *leader* to represent the parent node of the two nodes. For example, the mobot associated with the right sibling could always be chosen as the leader (unless there is no right mobot, in which case the left mobot is chosen). The leader node calculates the resulting product of the two values and saves this. The non-leader mobot stores its value as well (this value will be used during the down-sweep phase). The two mobots chosen as leaders at step i proceed in step $i + 1$ to communicate with each other, representing the internal nodes that are siblings in the tree. In this way, we continue marching up the implicitly-defined tree T computing the products of mobot descendents for each node v.

Note that at the end of the up-sweep phase, for each internal-node left child, v, there is a mobot sitting at a meeting point that stores the product of descendents of v.

The Down-sweep phase. During the down-sweep phase of the algorithm, we have each leader x store the product of preorder mobot descendents of the node v in T it is representing, and we ask x to retrace its steps. When a mobot x returns to meet with a mobot y, x knows the product of preorder leaf predecessors of y's node, u. So x tells y this value, which takes at most $O(w)$ time. Likewise, y tells x (again) the value of y's product of mobot descendents. The mobot x then multiplies this value to its current value of preorder leaf predecessors, which gives the correct value for u's sibling, which now becomes the node of T associated with x. This gives x the information it needs to then proceed down T to the next level, continuing

the down-sweep phase.

There are $O(\log N + \log M)$ rounds in the above algorithm, each of which takes $O(w)$ steps. In addition, any mobot travels distance at most $O(N)$ during all the horizontal rounds (since that side-length is doubling with each such round) and distance at most $O(M)$ in all the vertical rounds (since that side-length is doubling with each such round). Thus, we have the following.

Theorem 1. *Given n mobots in an $N \times M$ grid, the prefix products problem can be solved in $O((\log N + \log M)w)$ steps and with maximum distance traveled $O(N+M)$ in the long-haul PMR model with word-size w, in a parking-free fashion.*

We note that the above distance bound is optimal in the worst case. That is, a mobot may have to traverse a distance of $O(N + M)$ just to meet another mobot. Moreover, the above theorem implies an algorithm in the unit-cost model that runs in $O(N + M)$ time. We can, however, do better than Theorem 1 for the rectangle ranking problem, as well as many useful examples of the $\otimes$ operator, as we show next.

2.2 Improved Rectangle Ranking and Prefix Sums

For the rectangle ranking and prefix sums problems, we can significantly improve Theorem 1. Rectangle ranking and prefix sums are special cases of prefix products where $\otimes$ is addition. In the rectangle ranking problem we also have the additional constraint that every initial value stored in a mobot is 1. Thus, for this problem, w is $O(\log n)$, which is $O(\log N + \log M)$. Likewise, we expect w to be $O(\log N + \log M)$ for most applications of prefix sums as well.

In this section, we describe how to solve the prefix sums and rectangle ranking problems in $O(w + \log N + \log M)$ time in the long-haul PMR model with maximum distance $O(N + M)$. The main idea of our improved algorithm is to first bring together groups of $O(w)$ mobots, which then can act in concert to perform a series of pipelined summations.

Let us describe then how we can bring together groups of $O(w)$ mobots. We begin by partitioning the $N \times M$ grid into subgrids of size $O(w + \log N + \log M)$ and compressing all the mobots in each subgrid in $O(w + \log N + \log M)$ using unit-cost steps. By starting from these subgrids, we can assume that each internal node in T has a (physical) queue of size $O(w)$ associated with it, while keeping the overall rectangle dimensions at $O(N) \times O(M)$.

We then perform $O(\log N + \log M + w)$ steps of an attempt to route mobots towards the root of T. When two mobots are the first to meet at a node v of T, one of them stays behind and the other continues up the tree T. The mobot that stays behind is called the ***bouncer***. It is the job of the bouncer to keep a count of the number of mobots that pass through v so that it can inform new mobots that come to v whether or not they can join one of two queues associated with v—a left queue and a right queue. A mobot adds itself to a queue by first getting approval from the bouncer and then marching next to the queue from the front until it finds the free spot, where it inserts itself. Such a protocol allows us to pipeline queue insertions in $O(1)$ time per insertion. The bouncer adds mobots coming from v's left child to v's left queue and mobots from v's right child to v's right queue. The bouncer continues to allow queue insertions until it sees a total of w mobots, at which point we say that v is ***completed***. When v is completed, the bouncer at v informs any other mobots wishing to join v's queues that they are not allowed. A bouncer continues to allow mobots to leave its queues until the bouncer at v's parent, u, rejects insertions to its queues. The bouncer always gives precedence to the mobots in v's left queue, however, over its right queue. That is, with each step that one of v's queues is nonempty, the bouncer for v lets a mobot x leave a queue for v and try to move to u's queue, giving priority to mobots in the left queue. If x is rejected by u's bouncer, then it returns to v immediately, and the bouncer for v stops allowing mobots to travel up to u. If x is not rejected by u's bouncer, then it joins one of u's queues. Likewise, if v's left queue is empty, then the bouncer allows a mobot x to proceed similarly to u from v's right queue. Thus, mobots continue to move up according to their inorder relationships and be queued at nodes in the tree T until reaching the root or a completed node v.

Note that if a node becomes completed, then all its ancestors will become completed as well, using only $O(w + \log N + \log M)$ steps. After this computation is done, we run it in reverse and let the mobots repeat their actions backwards, which they can remember using an internal memory of size $O(w + \log N + \log M)$. But rather than having mobots return all the way to the leaves of T, we have them stop at the first uncompleted node on the path from T's root to their leaf starting point. Each such uncompleted node v can have at most w mobots return to it; hence, all of them can fit in v's queue. Moreover, by our priority scheme, the mobots in v's queue are sorted in reverse inorder. Thus, we can reverse this ordering in $O(w)$ time to sort each group of mobots according to the inorder of their starting point in T, that is, according to their lexicographic order in the starting rectangle.

At this point, we have all the mobots distributed into contiguous linear queues of size $O(w)$ in their correct orders. Let $\{a_1, a_2, \ldots, a_k\}$ denote the values stored in these mobots, in lexicographic order. In each group, we now initiate a pipelined prefix sum computation, where, starting with least-significant bits, we compute $a_1 + a_2$, $a_1 + a_2 + a_3$, …, $a_1 + \cdots + a_k$. Each time a mobot determines a bit of its prefix sum, it communicates that bit with its next-door neighbor. Thus, since it takes $O(w)$ steps to compute the next prefix sum given the previous one, and there are $O(w)$ mobots in the group, this pipelining allows each mobot in such a group to compute its associated prefix sum in a total of $O(w)$ steps.

At this point, we have computed a prefix sum in each group of $O(w)$ mobots associated with a node v in T that is uncompleted, but which is the child of a completed node. The remainder of our computation is to perform the up-sweep and down-sweep phases in T in a similar pipelined fashion to how we computed the prefix sums in each group. The main challenge is that the nodes in T are not right next to each other, as was the case of the mobots in each group.

We now are ready to perform the up-sweep and down-sweep phases in T. We elect the last member of each group as the representative of that group, sending that mobot to the parent node in T to communicate the sum of values from this group and perform the addition of this value with the sum from the sibling node in T. Mobots from each group also begin a routing to the root. But now the job of the mobots not computing sums of two children values is to communicate bits from a child to a parent. That is, as each bit is computed in a pipelined addition at a node v, there is a mobot that carries that bit to the parent so that it can be used to perform the next binary addition at the parent, and so on. After $O(w + \log N + \log M)$ steps the up-sweep phase will be done. A similar computation in reverse performs the down-sweep phase (stopping at the nodes just below completed nodes). Moreover, each mobot is guaranteed to traverse each edge of T at most $O(1)$ times. Thus, the total distance traveled is $O(N + M)$. Therefore, we have the following:

Theorem 2. *Given n mobots in an $N \times M$ grid, with each mobot storing a numerical value represented with w bits, we can compute all the prefix sums of these values in $O(w + \log N + \log M)$ time in the long-haul model in a parking-free fashion with maximum distance $O(N + M)$.*

Improved Prefix Products for NC Functions. An n-input Boolean function f is in **NC** if it can be computed in polylogarithmic depth by a logspace-uniform Boolean circuit of polynomial size (e.g., see [13]). If we assume that such a circuit is represented in a layered fashion, it can easily be simulated by a polynomial number of mobots in polylogarithmic time with polynomial maximum distance. We can use this fact to improve our prefix products algorithm for associative operators that can be computed in **NC** such that there is an upper bound w on the input and output size (corresponding to the word size for the values in the prefix products problem). For example, modular multiplication is such a problem.

Let $\otimes$ now be an associative operator that can be computed in **NC**, that is, by a bounded fan-in circuit of size $O(n^k)$ and depth $O(\log^k n)$, for constant $k \geq 0$. To improve our prefix products algorithm in this scenario, we perform a preprocessing step, where we do a rectangle ranking and compress the mobots into an $O(n^{1/2}) \times O(n^{1/2})$ grid, maintaining their lexicographic ordering, as either N or M must be at least $n^{1/2}$. This implies that the tree T is a balanced binary tree.

We then divide the computation of the up-sweep and down-sweep phases into two subphases each. Let

us focus on the up-sweep phase, as the down-sweep phase is similar. In the first subphase, we solve the prefix product problem up the tree T using the algorithm of Section 2.1, but we stop when we reach a level in T such that each node z on that level has $\Theta(w^k)$ descendents, where k is the constant for the size of the circuit needed to perform $\otimes$. Thus, this subphase takes $O(w \log w)$ time, since we do not count the time for performing the $\otimes$ operation inside a single mobot on two of its internal values.

Once the first subphase is complete, we will have completed the computation of the up-sweep phase up to each node z in T with $O(w^k)$ mobot descendents. At this point, we change modes and now perform the remainder of the computations up T using groups of $O(w^k)$ mobots to simulate the **NC** circuit to compute $\otimes$ at each remaining level. Since the inputs to these circuits can be computed in $O(1)$ time in parallel, this implies that we can perform each remaining $\otimes$ operation in $O(\log^k w)$ time, by simulating the **NC** circuit with each group of $O(w^k)$ mobots. Thus, the entire subphase can be implemented in $O(\log n \log^k w)$ time. Therefore, we have the following:

Theorem 3. *Let $\otimes$ be an associative operator that can be computed in **NC** with a (layered) circuit of size $O(w^k)$ in $O(\log^k w)$ time, such that there is an upper bound w on its input and output size (corresponding to the word size for the values in the prefix products problem). Suppose further that a value in the domain of $\otimes$ is stored in each of n mobots in an $N \times M$ grid. Then in $O(w \log w + \log N + \log M + \log n \log^k w)$ time we can solve the prefix products problem for $\otimes$ in the long-haul PMR model in a parking-free fashion.*

3 Coordinated Motion

Let us consider the problem of solving a coordinated motion planning problem, where we have n mobots, with initial positions, and each would like to move to a different, distinct position in a collision-free fashion. Naturally, we would like to perform this motion as quickly as possible. Note that we can reduce this problem to that of routing a permutation of n mobots that are initially placed in an $O(\sqrt{n}) \times O(\sqrt{n})$ grid (say, after rectangle-ranking and compression steps). In addition to the need to perform coordinated motion in any distributed robot swarm, this problem is also motivated by the fact that we can simulate each step of a bit-EREW PRAM algorithm by having one group of mobots represent memory cells and another group represent processors, with a permutation routing done between the two to perform the reads and writes.

3.1 Grid Lock with the Greedy Algorithm

As in the mesh-computer model (e.g., see [17], the greedy algorithm for permutation mobot routing is to have each mobot first move to its desired column and then move to its desired row. In the mesh-computer model, it is easy to see that this algorithm will route in $O(\sqrt{n})$ steps with queues of size $O(\sqrt{n})$. For mobots, however, the algorithm is potentially quite inefficient.

Theorem 4. *There is a permutation that causes the greedy mobot algorithm for n mobots in an $O(\sqrt{n}) \times O(\sqrt{n})$ grid to take $\Theta(n)$ time.*

Proof. Assign Cartesian coordinates to the locations so that all mobots have their destination (x, y) address greater than their source address. WLOG, let us assume that they all attempt to move leftwards to the correct y-address, and then turn onto a vertical track to move upwards to the correct x-address, but are blocked from turning when the vertical track is occupied. We'll also assume straight-going traffic has priority over turning traffic. We fix a parameter $k \approx \sqrt{n}/4$. We place an initial set of $k(k+1)$ mobots at (x, y) for $0 \leq x \leq k$ and $0 \leq y < k$. The mobots at (k, y) go to $(k, 2k + y)$. The mobots at (x, y) for $x < k - y$ go to $(k + y, k + x)$. The remaining mobots at (x, y) go to $(2k - y - 1, 3k + x)$. So, each row y has a train of mobots heading for column $k + y$, together with one mobot in each earlier column. Initially, the mobots in column k are blocking all the other mobots: they are moving vertically in column k and each other row starts with a mobot that wants to move into column k. After the first step, these mobots are followed into column k by the train of mobots in row 0, and all the other rows stay frozen. Only after the end of this train passes row 1, after

k steps, can the first mobot in row 1 move into column k. This frees the train of mobots in row 1 to start moving into column $k+1$, which they do just in time to block each higher row, and so on. □

3.2 Constant-Time Deterministic Long-Haul Routing

In this section, we show that the permutation routing problem can be solved in constant time in the long-haul model with an efficient bound on the maximum distance traveled. Before we give this algorithm, however, observe that if we have n mobots lined up compactly in a single row (or column), then we can easily route any permutation in $O(1)$ time with maximum distance $O(n)$, simply by having all the mobots move to their destinations, first with those wishing to move left and then with those wishing to move right. Such an approach uses parking, of course, but we can achieve a similar result without parking by having the mobots move on different tracks based on their direction and desired distance they wish to move.

The Spread-and-Compress Algorithm. We can achieve an improved bound for maximum distance traveled for constant-time routing, however, by being more clever in how we organize movements. In particular, we can achieve constant-time routing with $O(n^{2/3})$ maximum distance, which we conjecture is optimal for constant-time routing.

Let us assume that the mobots are initially configured in an $n^{1/2} \times n^{1/2}$ grid (we omit floor and ceiling notation to keep the discussion simple). Using a vertical and horizontal move, spread these out to an $n^{2/3} \times n^{2/3}$ grid, so that there are $n^{1/6}$ empty rows and columns between mobot positions. More to the point, next to each mobot x there is an empty $n^{1/6} \times n^{1/6}$ subgrid, $G(x)$, which contains $n^{1/3}$ cells. Divide the entire grid into vertical slabs of dimension $n^{1/3} \times n^{2/3}$, each of which contain mobots in an expanded $n^{1/6} \times n^{1/2}$ grid. Number the slabs left to right from 1 to $n^{1/3}$. Each mobot x has a current column location j in its slab and a slab k that it wishes to move to. For each slab i, move each mobot x in that slab to position i in $G(x)$ (using a row-major numbering). (See Figure 1a.) Next, have each mobot move to position i in $G'(x)$, which is a translation of $G(x)$ to location j in slab k. This is a horizontal movement for each mobot. Moreover, since there are at most $n^{1/3}$ mobots that may want to move to position j in slab k, each coming from a different slab, there are no conflicts in this greedy movement of mobots to their desired slabs. (See Figure 1b.) Of course, it is possible to have a completely full subgrid $G'(x)$, if all the mobots in a row wish to move to the same slab.

At this point, each mobot is in its desired destination slab, which contains $n^{1/6}$ destination columns of size $n^{1/2}$ each, but it still may not be in its desired column or row. Nevertheless, each mobot is in a "mega" column of $n^{2/3}$ cells. We now view this mega-column as $n^{1/2}$ destination columns stacked on top of one another, from top to bottom. That is, each destination column is viewed as being replicated $n^{1/3}$ times, and all the copies of the same destination column are lined up next to each other horizontally. We then have each mobot move vertically in its mega-column to its desired destination column and row in that column. (See Figure 1c.) There will be no conflicts in this movement, since we are routing a permutation. Once that is done, we merge all the mega-columns in a slab into one mega-column. As with the previous movement, this movement will not cause any conflicts, since we are routing a permutation. (See Figure 1d.)

After this movement, each slab will be stacked up in a single mega-column. We then unstack this mega-column and compress the entire expanded grid back to an $n^{1/2} \times n^{1/2}$ grid to complete the process. Thus, we complete an arbitrary permutation of the mobots in a constant number of steps, with the total movement made by any mobot in the entire algorithm being $O(n^{2/3})$.

Theorem 5. *Given n mobots in an $n^{1/2} \times n^{1/2}$ grid, with unique grid destinations, each mobot can move in the long-haul PMR model to its destination in $O(1)$ steps with $O(n^{2/3})$ maximum distance.*

This algorithm is not parking-free, however. We describe such an algorithm next.

3.3 Randomized Long-Haul Routing

In this section, we show how to route a permutation among n mobots in an $n^{1/2} \times n^{1/2}$ grid in $O(\log n)$ time with maximum distance $O(n^{1/2})$, with high probability, in the long-haul PMR model. The main idea of our algorithm is to mimic the algorithm of Maggs and Sitaraman [18] for routing packets on a butterfly network. Their algorithm uses constant-size queues and routes k packets from each of N inputs to random destinations in $O(k + \log N)$ steps. Such an algorithm can be used for arbitrary routing using Valiant's idea [29] of routing first to random locations and then from there to their true destinations.

Our routing algorithm is based on applying these two ideas at a somewhat coarse-grained level, where we subdivide the grid into vertical slabs of width $O(\log n)$ and likewise into horizontal slabs of height $O(\log n)$. First we route each mobot to a random column slab and then to a random row slab. Then we route each mobot to its destination vertical slab, and from there to its destination horizontal slab (keeping in the destination vertical slab). From there we then route each mobot to its true destination using the unshuffle-route-reshuffle algorithm of the previous subsection (with each horizontal slab in a group of $O(\log n)$ slabs using a different vertical column for the unshuffle).

Emulating the Maggs-Sitaraman algorithm presents some interesting technical challenges:

1. We need to layout butterfly networks in each row and column using parking-free mobot long-haul tracks without increasing the $n^{1/2} \times n^{1/2}$ grid by more than a constant factor.
2. The Maggs-Sitaraman algorithm uses empty packets, which they call "tokens," to separate waves of actual packets. We need to emulate their algorithm using mobots, which, using their terminology, correspond only to actual packets, not tokens.
3. The Maggs-Sitaraman algorithm uses busy-waiting to test when a full queue becomes unfull. We must avoid busy-waiting, as it could negatively impact our maximum distance bound.
4. We need to show that applying Valiant's idea at the coarse-grain slab level allows us to route a permutation in $O(\log n)$ steps so that, for any destination slab row or slab column, the number of destinations in that row/column is $O(\log n)$ w.h.p. This allows us to expand the grid by only a constant factor, keeping the maximum distance traveled to be $O(n^{1/2})$.

We give the details in the full version how we overcome the above technical challenges. Let us nevertheless sketch the main ideas here. First, for the layout problem, we use the scheme illustrated in Figure 2, which includes the inputs and outputs. Since there are $N = O(n^{1/2}/\log n)$ slabs in any row, we can layout an N-node butterfly using $O(N \log N) = O(n^{1/2})$ cells, including constant-sized (physical) queues associated with each butterfly network input.

To emulate the Maggs-Sitaraman algorithm without tokens, we implement each routing in three phases. We number the mobots in a row/column that we wish to route. In Phase 1, each odd mobot will act as an

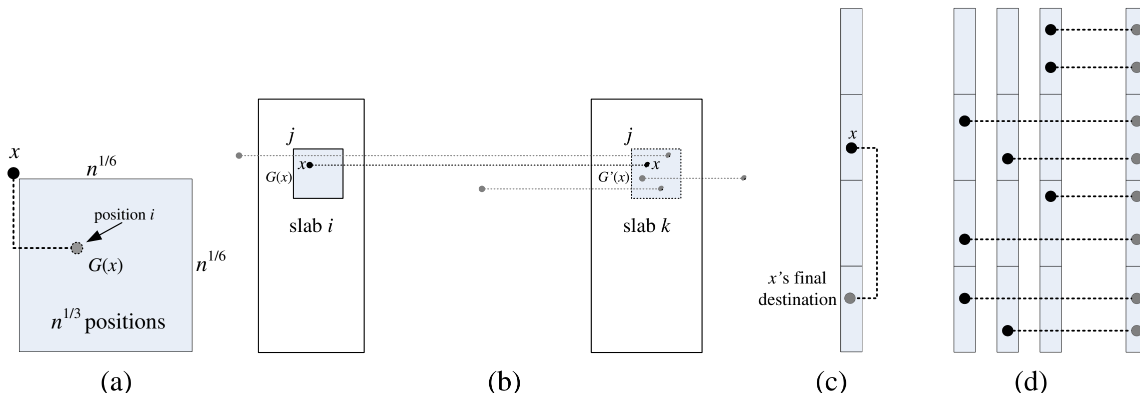

Figure 1: Constant-time routing. (a) moving to position i in a grid $G(x)$; (b) moving to slab k; (c) moving to the destination row-column copy in a mega-column; (d) merging mega-columns.

actual packet while each even mobot acts as a token. We then emulate the Maggs-Sitaraman algorithm, with all the mobots remembering the steps they took (which can be done using $O(\log n)$ space).

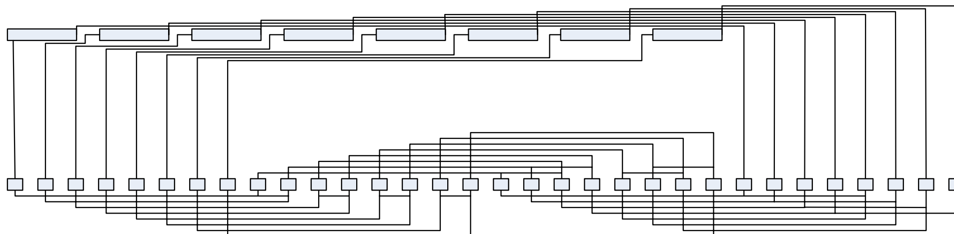

Figure 2: Layout out of the butterfly network using long-haul tracks.

We then undo this phase, as it is only a ***rehearsal*** (but the odd mobots must still continue to remember the steps they took in this phase). We then perform Phase 2, where the odd mobots act as tokens and the evens as real packets, with the odds again remembering the steps they take. After performing this phase, the even mobots are at their destinations. We then undo this phase for the odd mobots and, using the information saved from their rehearsal in Phase 1, we have them repeat that performance in Phase 3 to route themselves without conflicts to their destinations.

To avoid busy-waiting is actually simple. We use friendly waiting instead, so that once a queue becomes unfull, the last mobot in that queue returns to its previous node to tell the waiting mobot that it can now proceed. In this way, each mobot will traverse each edge back-and-forth at most twice during any phase; hence, the maximum distance traveled is equal to $O(n^{1/2})$ the width of a horizontal slab (or height of a vertical slab). Finally, we use a Chernoff bound to show that, with high probability, the number of mobots wishing to enter any row of a vertical slab or column of a horizontal slab is $O(\log n)$.

Theorem 6. *Given n mobots in an $n^{1/2} \times n^{1/2}$ grid with unique grid destinations, the mobots can move to their destinations in $O(\log n)$ time with maximum distance $O(n^{1/2})$, with high probability, in the long-haul PMR model, in a parking-free fashion.*

3.4 Routing in the Unit-Cost Model using Shrinking Squares

We can show that we can route n mobots in an $n^{1/2} \times n^{1/2}$ grid in $O(n^{1/2})$ time in the unit-cost model. We omit the details here, but the main idea is to emulate a mesh-computer sorting algorithm (e.g., see [17]) $O(\log n)$ times, once for each bit in the destination addresses, in a recursive fashion. To show that the total time is $O(n^{1/2})$ we observe that we can shrink subgrids by a constant factor with each recursive call.

4 Sorting

In the full version, we describe an algorithm that sorts n items, stored one per mobot, in $O(\log n)$ comparison rounds using n mobots in the long-haul PMR model. We assume that as input we are given n mobots packed in an $\sqrt{n} \times \sqrt{n}$ square. As the output, the algorithm will rearrange the mobots in a snake-like fashion in a square. Our algorithm is a simulation of Cole's EREW parallel sorting algorithm [11].

Performing this simulation in the long-haul PMR model presents a number of interesting challenges. For example, Cole's algorithm runs in $O(\log n)$ time by pipelining a series of parallel merges. We would like to likewise pipeline "living" merges, but do so faster than our general routing algorithms presented in the previous section. In addition, we have only n mobots to perform the work of $O(n \log n)$ comparisons, which presents an interesting assignment problem. Finally, creating many copies of individual values is trivial in the PRAM model, and this fact is used by Cole's algorithm. Copying values in the PMR model is complicated by the coherence problem presented when those values are carried by different mobots to different physical locations in the enviorment. We provide more of the details of how we deal with these challenges in the appendix.

References

[1] E. M. Arkin, M. A. Bender, S. P. Fekete, J. S. B. Mitchell, and M. Skutella. The freeze-tag problem: How to wake up a swarm of robots. *Algorithmica*, 46(2):193–221, 2006.

[2] H. Azzag, C. Guinot, and G. Venturini. How to Use Ants for Hierarchical Clustering. In *Ant Colony Optimization and Swarm Intelligence (Proc. 4th International Workshop, ANTS)*, volume 3172 of *Lecture Notes in Computer Science*, pages 350–357, 2004.

[3] O. B. Bayazit, J.-M. Lien, and N. M. Amato. Swarming behavior using probabilistic roadmap techniques. In E. Sahin and W. M. Spears, editors, *Swarm Robotics WS*, volume 3342 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2005.

[4] G. Beni and J. Wang. Theoretical problems for the realization of distributed roboticsystems. *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1914–1920, 1991.

[5] G. Beslon, F. Biennier, and B. Hirsbrunner. Multi-robot path-planning based on implicit cooperation in a robotic swarm. In *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, pages 39–46, New York, NY, USA, 1998. ACM Press.

[6] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford Univ. Press, 1999.

[7] R. A. Brooks. Intelligence Without Representation. *Artificial Intelligence*, 47:139–159, 1991.

[8] M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Solving the robots gathering problem. *Proc. 30th Int. Colloq. on Automata, Languages and Programming*, pages 1181–1196, 2003.

[9] R. Cohen and D. Peleg. Robot convergence via center-of-gravity algorithms. *Proc. 11 thInternational Colloquium On Structural Information And Communication Complexity (SIROCCO 11)*, pages 79–88, 2004.

[10] R. Cohen and D. Peleg. Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements. *Proc. 23 thAnnual Symposium on Theoretical Aspects of Computer Science (STACS '06)*, pages 549–560, 2006.

[11] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[12] T.-R. Hsiang, E. M. Arkin, M. A. Bender, S. P. Fekete, and J. S. Mitchell. Algorithms for rapidly dispersing robot swarms in unknown environments. In *Algorithmic Foundations of Robotics V*, pages 77–94. Springer, 2003.

[13] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. Elsevier/The MIT Press, Amsterdam, 1990.

[14] J. Kendall Preston and M. J. B. Duff. *Modern Cellular Automata: Theory and Applications*. Plenum Press, 1984.

[15] T. H. Labella, M. Dorigo, and J.-L. Deneubourg. Division of labor in a group of robots inspired by ants' foraging behavior. *ACM Trans. Auton. Adapt. Syst.*, 1(1):4–25, 2006.

[16] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. Assoc. Comput. Machinery*, 27(4):831–838, Oct. 1980.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1992.

[18] B. M. Maggs and R. K. Sitaraman. Simple algorithms for routing on butterfly networks with bounded queues. *SIAM J. Comput.*, 28(3):984–1003, 1999.

[19] J. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. *Robotics and Autonomous Systems*, 27(3):171–194, 1999.

[20] P. Rendell. Turing universality of the game of Life. In A. Adamatzky, editor, *Collision-Based Computing*, pages 513–539. Springer, 2001.

[21] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1997.

[22] S. Schockaert, M. D. Cock, C. Cornelis, and E. E. Kerre. Fuzzy Ant Based Clustering. In *Ant Colony Optimization and Swarm Intelligence (Proc. 4th International Workshop, ANTS)*, volume 3172 of *Lecture Notes in Computer Science*, pages 342–349, 2004.

[23] W.-M. Shen, P. Will, A. Galstyan, and C.-M. Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Auton. Robots*, 17(1):93–105, 2004.

[24] K. Støy, W.-M. Shen, and P. Will. Global Locomotion from Local Interaction in Self-recongurable Robots. In *Proceedings of the 7th international conference on intelligent autonomous systems (IAS-7)*, 2002.

[25] H. Tanner, A. Jadbabaie, and G. J. Pappas. Stable flocking of mobile agents, part I: Fixed topology. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, 2003.

[26] H. Tanner, A. Jadbabaie, and G. J. Pappas. Stable flocking of mobile agents, part II: Dynamic topology. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, 2003.

[27] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, 1987.

[28] X. Tu and D. Terzopoulos. Artificial Fishes: Physics, Locomotion, Perception, Behavior. *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.

[29] L. G. Valiant. A scheme for fast parallel communication. *SIAM J. Comput.*, 11:350–361, 1982.

[30] T. Wang and H. Zhang. Collective Sorting with Multiple Robots. In *Proceedings of First IEEE International Conference on Robotics and Biomimetics (ROBIO 2004)*, pages 716–720, 2004.

[31] S. Wolfram. *Cellular Automata and Complexity: Collected Papers*. Perseus Books Group, 1994.

A Sorting

In this appendix, we describe an algorithm that sorts n items, stored one per mobot, in $O(\log n)$ comparison rounds in the long-haul PMR model, where a comparison round involves the comparison of two items by two mobots. We assume that as input we are given n mobots packed in an $\sqrt{n} \times \sqrt{n}$ square. As the output, the algorithm will rearrange the mobots in a snake-like fashion in a square. Our algorithm is a simulation of Cole's EREW parallel sorting algorithm [11]. We assume in this appendix that the reader is familiar with the details of Cole's algorithm.

The area where sorting is conducted is partitioned into sub-areas in the following recursive manner. In the center of the plane a square of the total area of $O(n)$ is allocated. Therefore, the square dimensions are $O(\sqrt{n}) \times O(\sqrt{n})$. This square corresponds to the root of the computation tree in Cole's sorting algorithm and all computations associated with the root node are conducted in this area. On the left and right of the central square rectangular areas of the same height as the central square are allocated. These two rectangular areas correspond to the two children of the root node of the merge tree.

Above and below the rectangular spaces squares are allocated recursively to conduct all the calculations associated with sorting the items at the leaves of the subtrees rooted at the grandchildren of the root node (see Figure 3).

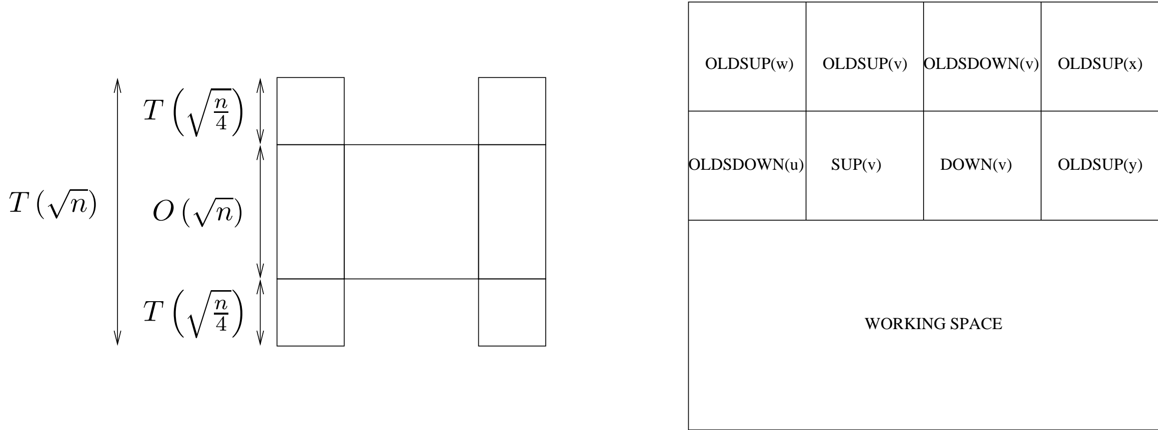

Figure 3: Recursive partitioning of the space for sorting using mobile robots and a zoom-in look at the subareas of the central square

The dimension of each side of the total space is determined by the following recurrence:

$$T(\sqrt{n}) = O(\sqrt{n}) + 2T(\sqrt{n}/2) = O(\sqrt{n}\log n)$$

Thus, the total space allocated has $O(\sqrt{n}\log n) \times O(\sqrt{n}\log n)$ dimensions.

To sort the input of n items in the PMR model, first, the mobots which store the inputs travel to the squares which correspond to the leaves of the computation tree. Each mobot can determine its destination coordinates by first conducting rectangle ranking as a preprocessing step of the algorithm and then calculating the exact coordinates, for a given layout.

In addition to each input mobot, we'll need some auxiliary mobots. In particular, in each space associated with a single node v of a tree we'll have mobots which represent the data of the following arrays SUP(v), SDOWN(v), OLDSUP(v), OLDSDOWN(v), OLDSUP(w), OLDSDOWN(u), OLDSUP(x) and OLDSUP(y) (see Figure 3 for an example of how these arrays might be stored). Note, that OLDSUP(w) $\cup$ OLDSDOWN(u) defines DOWN(v) and OLDSUP(x) $\cup$ OLDSUP(y) defines UP(v). At the end of each stage of the algorithm, the mobots associated with the older array data are replaced with mobots associated

with newly calculated arrays and the mobots being replaced will move up to areas associated with the parent nodes in the computation tree. Enough space is allocated from the begining to accomodate all the mobots of the auxilary arrays for every stage of the computation at that node. Since, the largest space required for each of the above arrays is at most $L(v)$, the total area allocated within each node is $O(L(v))$. We also allocate additional *working space* at each node. This working space is proportional to the size of the array L at that node. Thus, the total area still remains $O(L(v))$ at each node, which justifies the above recurrence equation.

Each of the mobots associated with the above arrays stores the corresponding rankings (a)–(j). During calculation of Steps 1 through 5, if a desired ranking is stored at a different node of the tree, the mobot travels to that node and receives the desired ranking by communicating with the corresponding mobot there. For example, in Step 1(ii), every mobot of OLDSUP(x) array at node v of the tree needs to travel to the corresponding mobot of the OLDSUP(x) array in the y node of the tree to obtain the OLDSUP(x) $\rightarrow$ SUP(y) ranking.

Once all the rankings required for the final calculation of a particular step are in place, the arrays from the neighboring nodes that need to be updated are moved to node v's working area. There the final cross-rankings are calculated according to the merging algorithm using at most 5 comparisons, that is, $O(1)$ comparison rounds. After the cross-rankings are calculated, the old arrays participating in cross-ranking are replaced with the new ones. The mobots corresponding to the respective old arrays at the children nodes are used as the new arrays. They are moved up to their parent node's working area, the values of the mobots in the working area are copied and they are moved to the appropriate area in the node. After this, the mobots in the working area return to their original locations.

Here are the detailed calculations of each step for node v. For simplicity we will say "array L is moving" to mean that mobots associated with array L are moving.

Step 1. Arrays OLDSUP(x) and OLDSUP(y) travel to appropriate sections of nodes x and y to receive the rankings described in Step 1(ii) through Step 1(v). Once those rankings have been updated, they travel together with the arrays SUP(x) and SUP(y) of nodes x and y to the working area of node v where the cross ranking SUP(x)$\times$SUP(y) are calculated. After the cross-rankings have been calculated, arrays OLDSUP(x) and OLDSUP(y) move to the working area of node u where they are joined by similar arrays from node w. There they communicate with newly cross-ranked arrays SUP(v) and SUP(w) and become exact copies of those arrays. The new copies of SUP(v) and SUP(w) travel to the location of old OLDSUP(v) and OLDSUP(w), which have been moved up themselves to the parent node of u. At this point, back at node v, the locations of OLDSUP(x) and OLDSUP(y) have been similarly filled with the appropriate arrays from nodes x and y now carrying new values of the cross-ranked arrays SUP(x) and SUP(y). At this point, arrays SUP(x) and SUP(y) in the working area return to their original position in nodes x and y.

Step 2. Arrays OLDSUP(w) and OLDSDOWN(u) travel to appropriate sections of nodes u and w to receive the rankings described in Step 2(ii) through Step 2(v). Once those rankings have been updated, they travel together with the arrays SUP(w) and SDOWN(u) of nodes w and u, respectively, to the working area of node v where the cross ranking SUP(w) $\times$ SDOWN(u) are calculated. After the cross-rankings have been calculated, arrays OLDSUP(w) and OLDSDOWN(u) move to the working area of node u where they are joined by similar arrays from node w. There they communicate with newly cross-ranked arrays SUP(w) and SDOWN(u) and become exact copies of those arrays. The new copies of SUP(w) and SDOWN(u) travel to the location of old OLDSUP(w) and OLDSDOWN(u), which have been moved up themselves to the parent node of u. At this point, back at node v, the locations of OLDSUP(w) and OLDSDOWN(u) have been similarly filled with the appropriate arrays from nodes x and y now carrying new values of the cross-ranked arrays SUP(w) and SDOWN(u). At this point, arrays SUP(w) and SDOWN(u) in the working area return to their original position in nodes w and u.

Step 3. At the beginning of Step 3, arrays SUP(v) and SDOWN(v) move to the location of arrays OLDSUP(v) and OLDSDOWN(v), respectively, while arrays OLDSUP(v) and OLDSDOWN(v) are no

longer needed and, therefore, they are moved up to node u to the locations of arrays SUP(u) and SDOWN(u). There, they are joined with the appropriate arrays from node w. The values of these mobots are reset and they will be used to create arrays NEWSDOWN(u) and NEWSUP(u). Similarly, arrays OLDSUP(x) and OLDSUP(y) move up to the (now vacant) location of SUP(v); while arrays OLDSDOWN(x) and OLDSDOWN(y) move up to (now vacant) location of SDOWN(v). The values of the mobots of these arrays will be reset and will be populated with values of arrays NEWSUP(v) and NEWSDOWN(v) during Step 3.

Arrays SUP(v), SUP(w), and SDOWN(u) (keeping in mind the new locations of these arrays) travel to the appropriate sections of nodes u and w to receive the rankings described in Step 3(ii). Arrays SUP(x), SUP(y) and SDOWN(v) travel to the appropriate sections of nodes x and y to receive the rankings described in Step 3(iii). At the end of each of the substeps the values of the arrays NEWSDOWN(v) and NEWSUP(v) are also populated. The rest of the rankings are already at the appropriate arrays at node v. At this point, arrays SUP(v), SDOWN(v), NEWSDOWN(v) and NEWSUP(v) travel to the working area of node v where the cross-ranking NEWSUP$(v) \times$ NEWSDOWN(v) are calculated. Once the arrays have been cross-ranked, all the arrays in the working area return to their appropriate locations.

For *Steps* 4 and 5 all the rankings of all required arrays have already been calculated from previous steps and are already at node v. So the only thing left to do is for the required arrays to travel to the working area and calculated the desired cross-rankings NEWUP$(v) \times$ NEWSDOWN(v) and NEWSUP$(v) \times$ NEWDOWN(v). After which they return to their original locations.