August 1986 Report No. STAN-CS-86-1129

# Beta Operations: Efficient Implementation of a Primitive Parallel Operation

by

Evan R. Cohn and Ramsey W. Haddad

**Department of Computer Science**

Stanford University
Stanford, CA 94305

# Beta Operations: Efficient implementation of a primitive parallel operation

Evan R. Cohn' — Stanford University
Ramsey W. Haddad — Stanford University

Abstract

We will consider the primitive parallel operation of the Connection Machine, the Beta Operation. Let the input size of the problem be $N$ and output size $M$. We will show how to perform the Beta Operation on an N-node hypercube in $O(\log N + \log^2 M)$ time. For a $\sqrt{N} \times \sqrt{N}$ mesh-of-trees, we require $O(\log N + \sqrt{M})$ time.

## 1 Introduction

The ever decreasing cost of computer processors has created a great interest in multi-processor computers. However, along with the increased power that this parallelism brings, comes increased complexity in programming.

One approach to lessening this complexity is to provide the programmer with general purpose parallel primitives that shield him from the structure of the underlying machine. In ***The Connection Machine*** [Hi85], Hillis suggests the ***Beta Operation*** as a parallel primitive for his hypercube-based machine. In this paper we shall explore efficient ways to perform this operator on several different well known architectures including the hypercube. We then present some lower bounds associated with the problem.

### 1.1 The Beta Operation

For a two-argument function, $F$, and an array of values, $C = [c_0, \ldots, c_n]$, let ***us*** define the ***F-reduction of*** C as the natural (APL style) reduction, ***except***

- $F/[c_0] = c_0$;
- $F/[c_0, \ldots, c_m] = F(F/[c_{\pi_0}, \ldots, c_{\pi_i}], F/[c_{\pi_{i+1}}, \ldots, c_{\pi_m}])$ for some $0 \leq i < m$, and some permutation $\pi$.

We are given an $F$, and N pairs, $(g_0, v_0), \ldots, (g_{N-1}, v_{N-1})$, as input to a Beta Operation. The $g_j$'s should be thought of as ***group numbers*** and the $v_j$'s as data. Let us call the collection of (g, v)-pairs with the value g = i, the i-block. Occasionally, we will also use the term i-block to refer to the set of processors holding the (g, v)-pairs of an i-block when no misunderstanding can result. Let $G = \{i \mid \exists j, g_j = i\}$ (that is, G is the set of the i's such that block $i$ has at least one element). If $s_i$ is the array of v-values from the i-block then a Beta Operation computes the values $y_i = F/s_i$, for $i \in G$.

Note that with our definition of the F-reduction of a block, performing a Beta Operation is ill-defined unless $F$ is commutative and associative.

There are two slightly different formulations of the Beta Operation. In both formulations, each processor j initially contains a pair, $(g_j, v_j)$. In the first formulation, the $|G|$ non-trivial $(i, y_i)$ pairs end up in sorted order (according to i) in the first $|G|$ processors. In the second formulation, at the end of the operation, each $y_i$ appears at processor i.

This difference is generally unimportant since output of the first type can be converted to that of the second type output with a single monotone routing. For all the networks that we'll consider, the time to perform a Beta Operation will dominate the time of a monotone routing.

## 2 Hypercube

We focus first on N processor hypercube systems, where there is a known bound on $|G|$. We shall discuss the necessary modifications for the case when $|G|$ is unknown in Section 4. For simplicity, we shall assume that $|G|$ is a power of 2. It is true that $2^{i-1} < |G| \leq 2^i$ for some i. If we assume that $|G|$ is really $2^i$, the algorithm will work with the same asymptotic time complexity. Let $N = 2^n$ and $|G| = 2^q$.[1]

### 2.1 The Generic Step

In each step of this algorithm, we will conceptually break the hypercube into smaller hypercubes. We then perform the Beta Operation on all of the

[1]The special cases where $q$ doesn't divide $n$ evenly or $|G|^3 > N$ can be treated by trivially modifying the algorithm given.

subcubes in parallel by applying the following sequence of subroutines.

**Sort.** We sort the (g, w)-pairs in the subcube by g-value.

**Reduce.** For each distinct g, we combine the pairs with that g into a single (g, $v$)-pair, by applying the function $F$ to the associated w-values.

**Compact.** We route the resulting (g, $v$)-pairs ($\leq |G|$ of them) into the lowest numbered processors of the subcube, retaining their sorted-by-g order.

We will organize the algorithm such that at the end of step $i$ in Phase 2, there will be (g, $v$)-pairs only in the $N/|G|^{i+2}$ processors with binary representations:

$$\overbrace{0\cdots0}^{(i+2)q}\overbrace{*\cdots*}^{n-(i+2)q}\ .$$

By the end of the last step ($i$ = $n/q$-3) of this phase, we will have performed the Beta Operation on the whole hypercube.

## 2.2 Phase 1

We break the $N$ processors into $N/|G|^3$ hypercubes of $|G|^3$ nodes each such that hypercube j has binary representation:

$$\overbrace{*\cdots*}^{3q}\overbrace{\ \ j\ \ }^{n-3q}\ .$$

For each hypercube we perform the following:

**Sort.** We use the odd-even merge sort to sort by g-value.

**Reduce.** Using O(log $|G|$) distribution-from-leaders [U84], we can combine the $|G|^3$ (g, $v$)-pairs into one (g, w)-pair per distinct g. Since this reduction takes the same time as the above sorting subroutine, $O(\log^2 |G|)$, it suffices for asymptotic analysis. Nevertheless, for various reasons that will become clear later, it is important to decrease the time taken by this step to O(log $|G|$). The reduction can be done efficiently as follows.

### 2.2.1 Efficient Hypercube Reduction

Let us call the largest hypercube contained in an i-block the central block (CB). (If an i-block has two largest hypercubes that can't be merged because their addresses are of the form $(k)\cdot 1 \bullet *j$ and $(k+1)\cdot 0\cdot *^{j}$, then always choose the lower numbered one.) We show in Lemma 2.1 below that the CB for an i-block of size $s$, $2^{j+2} - 1 > s > 2^{j+1} - 2$, must be of size $2^{j}$ or $2^{j+1}$. The reduction takes 3 steps.

**Step 1.** All the processors in a particular i-block determine if they are part of the CB for that i-block or not.

**Step 2.** All the processors, $p_k$, not in the CB for their i-block, send their (g, w)-pairs to either $p_{k+|CB|}$, $p_{k-|CB|}$, or $p_{k-2\cdot|CB|}$ depending on which of these is an address in their i-block's CB.

**Step 3.** In parallel, each of the CB's reduces all of its (g, $v$)-pairs to a single value.

In Step 1, each processor checks the two processors on either side to determine if it is the first or last processor in its i-block. Then, with two distribution-from-leaders, each processor can be told the numbers of the first and last processors in its i-block. Using this information a processor can determine if it is part of the CB in constant time. Step 2 is exectued as follows. There are at most $|CB| - 1$ elements in the $i$-block before the first element in the CB. If this were not the case then the first $|CB|$ elements would constitute the CB. There are at most $2 \cdot |CB| - 1$ elements after the CB, or the CB would be twice as big. The processors who are not part of the CB can send their pairs over to the appropriate processors of the CB with three montone routing steps. Step 3 is straightforward. The total time for all three steps is $O(\log N)$.

**Lemma 2.1** ***The CB for an i-block of size s,*** $2^{j+2} - 1 > s > 2^{j+1} - 2$, ***must be of size*** $2^{j}$ ***or*** $2^{j+1}$.

**Proof:** Clearly $|$ CB) can be no bigger than $2^{j+1}$.

Assume that we have a CB of size $2^{h}$, where $0 \leq h \leq j - 1$, with addresses of the form $(k)\ .\ *^{h}$, with $k$ even. The element at location $(k - 1)\ .\ 0^{h}$ is not in the i-block, because then the hypercube starting at *this* address would be the CB. The element at location $(k+1)\cdot 1^h$ is not in the i-block, because otherwise concatenating the blocks $(k) . *^h$ and $(k+1) . *^h$ would give us a CB of size $2^{h+1}$. Thus $s \leq 3 . 2^h - 2$, which is a contradiction for all $h$'s in the range specified.

Simlarly, we get a contradiction if we assume that we have a CB of size $2^h$ with addresses of the form $(k) . *^h$, with $k$ odd. ∎

**Compact.** Consider the (g, $v$)-pairs left by the reduce stage. By means of a prefix operation we can compute, for each (g, $v$)-pair, how many (g, $v$)-pairs are in lower numbered processors. Then we can compact via a monotone routing.

## 2.3 Phase 2

Steps i = 1 through i = $n/q - 3$:

We break the $N/|G|^{i-1}$ lowest numbered processors into $N/|G|^{i+3}$ hypercubes of $|G|^4$ nodes each, such that hypercube j has binary representation:

$$\overbrace{0\cdots 0}^{(i-1)q}\overbrace{*\cdots *}^{4q}\overbrace{\quad j \quad}^{n-(i+3)q} .$$

Note that with this choice of partitioning the hypercube, each subcube has only $|G|^2$ $(g,v)$-pairs. This is because before Step $i$ only the processors with addresses of the form:

$$\overbrace{0\cdots 0}^{(i+1)q}\overbrace{*\cdots *}^{2q}\overbrace{\quad j \quad}^{n-(i+3)q} .$$

contain (g, $v$)-pairs. For each subcube, perform the following:

**Sort.** We can use the Nassimi-Sahni sort [NS82], to sort the $|G|^2$ (g, w)-pairs by g-value.

**Reduce.** It is easiest to view the $|G|^4$ node hypercube as a $|G|^2$ x $|G|^2$ matrix, $p_{ij}$, with the processors arrayed in order of increasing processor number (in row-major form). Initially, only the first row contains (g, $v$)-pairs. Using a prefix operation, we can determine which processors are the leftmost processors in their i-block. Call these the *leaders*. We start by broadcasting the contents of each first row processor, $p_{1j}$, to the column j. Then each processor, $p_{jj}$, broadcasts to row j. Finally, in the columns of the leaders, $F$ is applied to those v-values whose corresponding g-values match the leader's.

Compact. As in Phase 1, consider the $(g, v)$ pairs left by the reduce stage. Move all of these pairs to the first row. All the processors that ***don't*** contain one of these pairs set their g-value to infinite. We can then compact by sorting on g-value using the Nassimi-Sahni sort.

## 2.4 Time Analysis

The sort step of Phase 1 takes $O(\log^2 |G|)$ time. The reduce and compact subroutines of Phase 1 both take $O(\log |G|)$ time. In every step of Phase 2, each of these subroutines takes $O(\log |G|)$ time. There are $O(\log N / \log |G|)$ such steps so Phase 2 takes time $O(\log N)$. Thus, the overall time for the algorithm is $O(\log N + \log^2 |G|)$.

## 3 Mesh-of-Trees

We first note that the Beta Operation can be performed easily in time $O(\sqrt{N})$ on a $\sqrt{N} \times \sqrt{N}$, N-processor mesh system, even if $|G|$ is not known beforehand. This upper bound is tight since there is an obvious lower bound of $\Omega(\sqrt{N})$ time even when $|G|$ is given. In the case of mesh-of-trees (MOT) our results are for the $\sqrt{N} \times \sqrt{N}$, $O(N)$ processor MOT system where there is a known bound on $|G|$. [2]

### 3.1 The Generic Step

The generic steps in Phases 2 and 3 will be essentially the same as the generic step of the hypercube algorithm. The essential difference is that the size of the sub-MOT's we work on will grow each step. Remember that in the hypercube, for the steps in Phase 2, we were always working with sub-hypercubes of a single size ($|G|^4$ nodes each). Another minor difference is that each sort-reduce-compress step is preceded by a routing step.

We start Phase 1 by performing the Beta Operation on sub-MOT's with side $\sqrt{4|G|}$. In Phase 2 we increase the size of the sub-MOT's considered until the number of processors is equal to the square of the number of remaining $(g, v)$-pairs in each sub-MOT. In Phase 3 we can then quickly increase the sub-MOT size to $\sqrt{N} \times \sqrt{N}$.

[2] As was the case with the hypercube, we shall disregard the special cases when divisions, square roots and logarithms produce non-integral values. Although these cases present no special problems, dealing with them introduces needless clutter.

## 3.2 Phase 1

Break the N processors into $N/4|G|$ sub-MOT's with side $\sqrt{4|G|}$. We can perform the Beta Operation on these sub-MOT's using just the mesh connections. Note that this can be done without knowing $|G|$ beforehand. We simply sort on the g-values and reduce the resulting i-blocks to single values.

## 3.3 Phase 2

Steps $i = 1$ through $3q/2 - 1$:

**Route.** Immediately before Step i, the MOT is divided into $N/(4^i|G|)$ sub-MOT's with sides of length $\sqrt{4^i|G|}$. The first $\lceil\sqrt{|G|/4^i}\rceil$ rows of each such sub-MOT contain the $\leq |G|$ different $(g, v)$-pairs, compacted to the left. For convenience, these initial rows of the sub-MOT shall henceforth be called the non-trivial-part (NTP). We start step $i$ by conceptually clumping 4 contiguous sub-MOT's into a single square sub-MOT with twice the side length. We first shift up the NTP's of the two lower blocks so that they are contiguous to the NTP's of the upper blocks. This results in a sub-MOT with side $\sqrt{4^{i+1}|G|}$ having a NTP occupying the first $2\lceil\sqrt{|G|/4^i}\rceil$ rows.

**Sort.** We can then sort this new NTP using the odd-even merge sort outlined in Theorems 3.2 and 3.1.

**Reduce.** For each group number there are up to four different (g, w)-pairs. We can combine these to produce one (g, v) pair in O(log $|G|$).

**Compact.** All processors not holding one of these pairs set their g-values to infinite. We then sort again on group number so that the NTP is compacted in the first $|G|$ spaces (in the row-major sense).

At the end of this phase, we have $N/|G|^4$ sub-MOTs of side $|G|^2$, each with no more than $|G|$ non-trivial (g, $v$)-pairs.

## 3.4 Phase 3

Steps i = 1 through $\log(n/2q - 1)$

In each step $i$ we will increase the side of the sub-MOT from $|G|^{2^{i-1}+1}$ to $|G|^{2^i+1}$. The last step will leave us a single MOT with side $\sqrt{N}$. Notice that in each sub-MOT, the number of processors will always be equal to the square of the number of non-trivial (g, $v$)-pairs.

The route-sort-reduce-compact stages are performed in each sub-MOT as follows:

**Route.** At the beginning of step $i$ we have sub-MOT's of side $|G|^{2^{i-1}+1}$, each with no more than $|G|$ non-trivial (g, $v$)- pairs. We will conceptually clump $|G|^{2^i}$ of these sub-MOT's into sub-MOT's of side $|G|^{2^i+1}$. Consider each such sub-MOT of side $|G|^{2^i+1}$ as being composed of $|G|^{2^{i-1}}$ columns of width $|G|^{2^{i-1}+1}$. In the routing step we move the NTP's from all the sub-MOT's in each column into the controllers[3] in that column as a preliminary to sorting.

**Sort.** For each clump we have a sub-MOT of $|G|^{2^{i+1}+2}$ processors and $|G|^{2^i+1}$ non-trivial (g, $v$)-pairs. Thus we can sort within each clump using the standard MOT algorithm [U84].

**Reduce.** The reduce step looks very much like the standard MOT sorting algorithm. First, every controller checks to see if the group number it contains is the leftmost such group number. As with the hypercube algorithm we shall call such processors leaders. Next, each controller broadcasts its value to its entire row and column. Finally, in the columns of the leaders, $F$ is applied to the v-values whose corresponding g-values match that of the column's leader.

**Compact.** Another sort will then compact the values. It is assumed as always that non-leader controllers have infinite g-values.

## 3.5 Timing

We use the following theorems in analyzing the time required to perform the algorithm for the Beta Operation on the MOT.

**Lemma** *3.1* ***An arbitrary partial permutation routing of s elements that start and end on the leaves of a complete binary tree with m leaves can be performed in time*** $O(s + \log m)$.

**Proof:** Let $S_{lr}$ be the elements that need to be routed from the left half of the tree to the right half. Similarly define $S_{rl}$, $S_{ll}$ and $S_{rr}$. We

[3]We follow the lead of Ullman [U84] in viewing the root of the ith column tree and ith row tree as being a single node. We shall refer to this node as the ith controller.

can pipe the elements of $S_{lr}$ to their destinations in time $O(|S_{lr}| + \log m)$. Similarly, the elements of $S_{rl}$ can be routed in time $O(|S_{rl}| + \log m)$. To route the elements of $S_{ll}$, we actually break it into two consecutive routings. In the first, the elements are routed from their starting locations in the left half of the tree to locations on the right, and then in the second they are routed from the right half back to their destinations on the left. This takes time $O(|S_{ll}| + \log m)$. Similarly, the elements in $S_{rr}$ can be routed in time $O(|S_{rr}| + \log m)$. Since $s = |S_{rl}| + |S_{lr}| + |S_{ll}| + |S_{rr}|$, the overall routing can be done in time $O(s + \log m)$. ▮

**Lemma 3.2** ***Given an MOT of side m with all elements contained in the first $s$ rows. In time $O(s + \log m)$, we can achieve any permutation in which the elements' final destinations are also within the first s rows.***

**Proof:** Let $R_{i,j}$ be the row of the destination of the element that starts in row $i$, column j; similarly, $C_{i,j}$ is the column of the destination. We apply Lemma 3.1 three times. It's applied first to the columns, then to the rows and then to the columns of the MOT such that each element from (i, j) follows the permutations: $(i, j) \rightarrow (i + j \bmod m, j) \rightarrow (i + j \bmod m, C_{i,j}) \rightarrow (R_{i,j}, C_{i,j})$. Each of these three permutation operations can be performed in $O(s + \log m)$ time yielding the desired result. ▮

**Theorem 3.1** ***Consider a MOT with side m. Assume that it is divided vertically into two halves and that the first $s$ ($1 \leq s \leq m$) rows on the left side contain the sorted list, A, and the first $s$ rows on the right side contain the sorted list, B. Then we can merge these two lists, with the results ending up in the first $s$ rows, in time $T(s, m) = O(s + \log m \log 2s)$.***

**Proof:** This can be done with odd-even merge.

**Step 1.** In step 1 we separate out the odd-position ***A*'s** ($A_{odd}$) from the even-position ***A*'s** ($A_{even}$) so that $A_{odd}$ occupies the first s/2 rows and $A_{even}$ occupies the s/2 rows starting at row m/2. This can be done in the manner of Lemma 3.2 in time $c_1(s + \log m)$. Simultaneously, we separate the ***B*'s.**

**Step** 2. In step 2 we exchange the positions of $A_{odd}$ and $B_{even}$. This can also be done in time $c_2(s + \log m)$.

**Step** 3. We now want to merge lists that are stacked vertically. Consider the $m/2$ x $m/2$ square in the upper left. We separate out the even-position $B_{even}$ ( $B_{even_{even}}$) and the odd-position $B_{even}$ ($B_{even_{odd}}$). The even positioned $B_{even}$'s go on the left and the odd ones on the right. Simultaneously, separate the $B_{odd}$'s, the $A_{even}$'s and the $A_{odd}$'s. This can be done in time $c_3(s$ + log m).

**Step** *4.* We exchange $B_{even_{even}}$ and $A_{even_{odd}}$. Simultaneously, exchange $B_{odd_{odd}}$ and $A_{odd_{even}}$. This can be done in time $c_4(s$ + log m).

**Step** 5. We now have *4* sub-MOTs with side $m/2$ and *s/2* rows. Recursively merge these in time $T(s/2$, m/2) yielding the four lists $AB_{even_{even}}$, $AB_{even_{odd}}$, $AB_{odd_{even}}$, and $AB_{odd_{odd}}$.

**Step** *6.* We interleave $AB_{even_{even}}$ with $AB_{even_{odd}}$. By merely swapping adjacent list elements we are left with the sorted list $AB_{even}$ . Simultaneously, we can interleave $AB_{odd_{even}}$ with $AB_{odd_{odd}}$, yielding the sorted list $AB_{odd}$ after the value swapping. Lastly, interleave $AB_{even}$ with $AB_{odd}$ and do any needed value swapping. This can be done in time $c_5(s + \log m)$.

**Basis.** If s = 1 then we can sort in time $O(\log m)$.

**Induction step.** Let s = $s_0$. Let $c_6 = c_1+c_2+c_3+c_4+c_5$. Then $T(s_0,m) \leq T(s_0/2, m/2) + c_6(s_0 + \log m)$. Thus $T(s,m) = O(s + \log m \log 2s)$.

▮

**Theorem** *3.2 Consider a MOT with side m. Assume that there are O(ms) numbers in the first s rows. We can sort this list with the results ending up in the first s rows, in time $T(s, m) = O(s + \log m \log^2 2s)$.*

**Proof:** We use a merge sort. First divide the MOT into four sub-MOTs of side m/2. Using routings of the type in Lemma 3.1, distribute the numbers into the first s/2 rows of each sub-MOT. Recursively sort in these sub-MOTs in time $T(s/2, m/2)$. We then merge the four sorted lists together using the methods outlined in Theorem 3.1. Hence, $T(s,m) = T(s/2, m/2) + c(s + \log m \log 2s)$. Solving this recurrence yields the time bound claimed above. ▮

The first phase will take time $O(\sqrt{|G|})$. For step i of the second phase, the time is determined by the sorting. By application of Theorem 3.2 we can see that the second phase will take time

$$O(\sum_{i=1}^{3q/2} 2 \left\lceil \sqrt{|G|/4^i} \right\rceil \text{ t } \log \sqrt{4^{i+1}|G|} \log^2 \mathbf{2} \left\lceil \sqrt{|G|/4^i} \right\rceil)$$

$$= O(\sqrt{|G|} + \log^4 |G|) = O(\sqrt{|G|}).$$

The third phase will take time $O(\sum_{i=1}^{\log(n/2q)} \log |G|^{2^i+1})$ = O(n). Thus the overall time is O(log $N$ + $\sqrt{|G|}$).

## 4 Determining the Output Size

The time taken by the algorithms given above is a function of both the input size, N, and the output size, $|G|$ (for convenience let $M = |G|$). The algorithms that are given assume that $M$ is known. Thus the question arises, what do we do if we don't already know M?

For a large class of problems, and Beta Operations appear to be one of them, the problem of determining the output size, M, is essentially as complex as the problem of computing the output given the output size. While it would be possible to develop separate algorithms to determine the output size, we will exhibit below a general scheme that enables one to determine $M$ ***and*** solve the problem in time proportional to that required for solving the problem ***given M.***

### 4.1 Iterative Guessing

We will use a strategy of iterative guessing that depends on having an algorithm, call it Algorithm A, with the following properties:

- The running time is $t(N$, Q), where Q is a given bound on the output size.
- If Q $\geq M$, then the algorithm works correctly and produces the appropriate output of size $M$.
- If Q < $M$, then the algorithm can detect the error (within time $t(N,Q)$).

- $t(N, Q)$ is monotonically nondecreasing in Q.

(These restrictions can be relaxed, but they are sufficient for our purposes.)

Using Algorithm A, we can create a new algorithm, Algorithm B, that can solve the problem without knowing $M$ beforehand. Algorithm B will sequentially try the guesses $(g_1, g_2, \ldots)$. That is, it will first run Algorithm A with $Q = g_1$. If this first guess is too small, it runs it with $Q = g_2$, then $Q = g_3$, etc... until Algorithm A finally succeeds.

It is clear that an arbitrary choice of $g_i$'s will not result in an efficient algorithm. Let us presumptuously define an ***efficiently-convergent guess sequence*** and then justify the name. Let us denote the minimum output size possible for any input by $M_{\min}$. An efficiently-convergent guess sequence, $(g_0, g_1, \ldots)$ for the function $t(N, Q)$ is a sequence such that:

$$\begin{array}{l} g_0 = M_{\min} \\ c_1 t(N, g_{i-1}) \leq t(N, g_i) \leq c_2 t(N, g_{i-1}) \\ 1 < c_1 \leq c_2 \end{array}$$

where $c_1$ and $c_2$ are independent of i, but can be chosen in a fashion that depends on the sequence of $g_i$'s.

**Theorem 4.1** ***Assume that we are given an algorithm for 'problem $\mathcal{P}$ given M' that has all the properties enumerated above. Then if we are given a corresponding eficiently-convergent guess sequence, we can create an algorithm to solve '$\mathcal{P}$ not given M', in time $\tilde{t}(N, M)$ where $\tilde{t}(N, M) = \Theta(t(N, M))$.***

**Proof:** The given algorithm for '$\mathcal{P}$ given $M$' is our 'Algorithm A'. To solve the problem '$\mathcal{P}$ not given $M$', we run 'Algorithm B'. Let $g_s$ be the guess for Q on which the Algorithm A finally succeeds.

From the properties of Algorithm A, it follows that $g_{s-1} < M \leq g_s$. Hence,

$$\begin{aligned} t(N, g_s) &\leq c_2 t(N, g_{s-1}) \\ &\leq c_2 t(N, M). \end{aligned}$$

Also,

$$c_1 t(N, g_{i-1}) \leq t(N, g_i)$$

$$\begin{aligned}
\sum_{i=2}^{s} c_1 t(N, g_{i-1}) &\leq \sum_{i=2}^{s} t(N, g_i) \\
t(N, g_1) + \sum_{i=1}^{s-1} (c_1 - 1) t(N, g_i) &\leq t(N, g_s) \\
\sum_{i=1}^{s-1} t(N, g_i) &\leq \frac{1}{c_1 - 1} t(N, g_s) \\
\sum_{i=1}^{s} t(N, g_i) &\leq \frac{c_1}{c_1 - 1} t(N, g_s).
\end{aligned}$$

From the definition of our algorithm for '$\mathcal{P}$ not given $M$;

$$\begin{aligned}
\tilde{t}(N, M) &\leq \sum_{i=1}^{s} t(N, g_i) \\
&\leq \frac{c_1}{c_1 - 1} t(N, g_s) \\
&\leq \frac{c_1 c_2}{c_1 - 1} t(N, M).
\end{aligned}$$

Since it is trivially true that

$$\tilde{t}(N, M) \geq t(N, M)$$

it follows that $\tilde{t}(N, M) = \Theta(t(N, M))$. Note that the optimal choice of $c_1 = c_2 = 2$ yields a factor of 4 slowdown in the worst case. ▌

## 4.2 Application of Method

**Lemma 4.1** ***For $t(N,Q) = c(\log N + \log^2 Q)$, the guess sequence $g_0 = 1$, $g_i = 2^{\sqrt{(2^i-1)\log N}}$ for $i > 0$ is eficiently-convergent with $c_1 = c_2 = 2$.***

Since the algorithm described in Section 2 satisfies the properties enumerated in Section 4.1, the corollary below follows from the above lemma and Theorem 4.1.

**Corollary** ***4.1 Beta Functions on $|G|$ groups can be computed in time $O(\log N + \log^2 |G|)$ on a hypercube, without prior knowledge of $|G|$.***

**Lemma 4.2** ***For $t(N, Q) = c(\log N + \sqrt{Q})$, the guess sequence $g_0 = 1$, $g_i = ((2^i - 1)\log N)^2$ for $i > 0$ is eficiently-convergent with $c_1 = c_2 = 2$.***

Since the algorithm described in Section 3 satisfies the properties enumerated in Section 4.1, the corollary below follows from the above lemma and Theorem 4.1.

**Corollary 4.2** ***Beta Functions on $|G|$ groups can be computed in time $O(\log N + \sqrt{|G|})$ on a MOT, without prior knowledge of $|G|$.***

## 5 Lower Bounds

In this section, we will prove some lower bounds, given our formulation of the Beta Operation, and relate them to the areas and times for the algorithms and architectures discussed above. Note that while in the other sections of this paper we use the word model of computation, here we use the bit model of computation.

The input is N pairs of numbers, $(g_0, v_0), (g_1, v_1), \ldots, (g_{N-1}, v_{N-1})$, each of which is in the range 0 to $N - 1$. Let $G_i = \{v_j \mid g_j = i\}$. For all $i$ such that $|G_i| > 0$, we output $(i, y_i)$ in sorted order (according to i) where $y_i$ is the F-reduction of $G_i$. As above, let $G = \{i \mid |G_i| > 0\}$. Let $w_0$ be the smallest member of G; similarly, let $w_i$ ($i < |G|$) be the $(i + 1)$-th smallest member of G. Define $z_i = y_{w_i}$; that is, $z_i$ is the y-value of the $(i + 1)$-th output. We will refer to the the j-th bit of $z_i$ as $z_{i,j}$. (Similarly for the g's.)

### 5.1 A Lower Bound on Area

(The structure of this lower bound proof closely follows the one for sorting in [U84].) First we will show that

**Lemma 5.1** ***In a when- and where-determinate circuit that performs the Beta Operation correctly for any $|G|$, none of the output bits $z_{i,j}$ (for $i < N - 1$) can be output before all of the input bits $g_{i,j}$ (for $j > 0$) have been read.***

**Proof:** Assume, to the contrary, that $z_{p,q}$ ($p < N - 1$) is output before $g_{r,s}$ ($s > 0$) is read. We construct two possible inputs. Fix every $g$ and $v$, except $g_r$, as follows

- Choose a $t \neq r$. Set $g_t = p + 1$; $v_t = 2^q$.
- Set all other $v_i = 0$.

- For all $i$ (other than i = ♦ and $i = r$) choose a value of $g_i$ (different from $p$ and $p+1$) in such a way that $\forall j$, $0 \leq j < p$, $\exists i$ such that $g_i = j$.

The two possible inputs yielded by setting either g,. = $p$ or $g_r = p \oplus 2$" produce different values for the bit $z_{p,q}$. Yet $z_{p,q}$ is output before $g_{r,s}$ is read – contradiction. ∎

**Theorem 5.1** ***Any when- and where-determinate circuit that can perform a Beta Operation on N inputs must have A = $\Omega(N \log N)$.***

**Proof:** (This proof assumes N is even. The proof for N odd is similar.) We will construct a family of inputs of size *(N/2)!* each with different outputs. For all i, fix $v_i$ = i. For $i \geq N/2$, fix $g_i = 2i - N + 1$. **Now,** we allow the remaining inputs, go,. . . , $g_{N/2-1}$ to be any permutation of the even numbers less than N.

Focus on the time just before the first $z_{i,j}$ (for i < N − 1) is output. The circuit has already read all of the bits that differ between the elements of our family of inputs. Hence, all inputs read subsequently will be the same for any element of our family. Since the circuit must still produce (N/2)! different outputs, it must have at least (N/2)! states and hence at least log(( N/2)!) = $\Omega$( N log N) bits and area. ∎

## 5.2 An $AT^2$ Lower Bound

(The structure of this lower bound proof closely follows the one found in [Ho85].)

**Theorem** *5.2* ***Any when- and where-determinate circuit that can perform Beta Operations for any M = $|G|$ requires $AT^2 = \Omega(NM \log N)$, where N is the number of input pairs.***

**Proof:** If there is an input pad that reads $\Omega(M^{1/2})$ bits of the $v_i$'s then $T = \Omega(M^{1/2})$. This, coupled with the above theorem, implies our $AT^2$ bound.

If no pad reads $\Omega(M^{1/2})$ bits, then we may partition our circuit into a set of blocks $B$ with $|B| = \Theta(\frac{N \log N}{M})$ so that

- each block in $B$ has area $0(\frac{AM}{N \log N})$ and perimeter [4] $O(\sqrt{\frac{AM}{N \log N}})$;

***We use "perimeter" to mean the perimeter not including the external boundary of the circuit.**

- each block in $B$ reads at most O(M) bits of the $v_i$'s.

Such a collection of blocks is obtained by first cutting the circuit into $\Theta(\frac{N \log N}{M})$ blocks each of perimeter $O(\sqrt{\frac{AM}{N \log N}})$ (Lemma *2.3* of [U84]). Then another $\Theta(\frac{N \log N}{M})$ cuts suffice to ensure that each resulting block reads at most $I_{\max} = O(M)$ bits of the $v_i$'s.

Two statements are true of the above set of blocks:

- At least half of these blocks produce less than twice the average number, $\Theta(\frac{M^2}{N})$, of the output $z_i$ bits for $i < M$.

- Let $I_{\text{ave}}$ be defined as $N \log N/|B| = O(M)$. More than half of the blocks read at least $2I_{\text{ave}} - I_{\max}$ input bits.

Note that we can stay within our block cutting rules and still make our blocks small enough so that $2I_{\text{ave}} - I_{\max} = \Omega(M)$. Thus, there is some block, $b \in B$, that:

- reads at least $l_1 = \Omega(M)$ input bits from $l_2$ of the $v_i$'s (assume without loss of generality that these are from $v_0, \ldots, v_{l_2-1}$);

- has perimeter $O(\sqrt{\frac{AM}{N \log N}})$;

- produces $l_3 = O(\frac{M^2}{N})$ of the $z_i$ output bits with $i < M$.

We may then construct a fooling set as follows. Let

$$V = \{(i, j) \mid b \text{ reads in bit } j \text{ of } v_i\}$$

$$Z = \{i \mid b \text{ outputs a bit of } z_i, \text{ and } i < M\}.$$

For each $1 \leq i \leq l_2$, choose a. distinct $\alpha_i$ such that $\alpha_i \notin Z$ and $0 \leq \alpha_i < M$ (note that we can stay within our block cutting rules and still make our blocks small enough so that $l_2 + l_3 < M - 1$). Let $c = M - l_2$. Choose $\beta_1, \ldots, \beta_c$ (each $< M$) to be distinct from each other and the $\alpha_i$'s. Choose the $g_i$'s as follows

$$\begin{aligned} g_i &= \alpha_i, && \text{for } i = 1 \text{ to } l_2 \\ g_{i+l_2} &= \beta_i, && \text{for } i = 1 \text{ to } c \\ g_i &= \beta_1, && \text{for all other } i. \end{aligned}$$

And the $v_i$'s as

$$\text{bit } j \text{ of } v_i = 0 \text{ or } 1, \quad \text{for } (i, j) \in V$$
$$\text{bit } j \text{ of } v_i = 0, \quad \text{for } (i, j) \notin V$$

Each of our $2^{l_1}$ choices for the input yields a different configuration of the output bits outside of $b$. As $l_1$ = Q(M), the fooling set is of size $2^{\Omega(M)}$. This yields the bound $\sqrt{\frac{AM}{N \log N}}T$ = Q(M), that is, $AT^2 = \Omega(NM \log N)$. ▮

## 6 Conclusions

We showed in Section 2 that the Beta Function Problem can be solved in time O(logN + $\log^2 |G|$) on a hypercube. We can achieve this same time bound on a shuffle-exchange graph. In Section 3, we showed that the problem can be solved in time 0( $\sqrt{|G|}$ + log N) on a mesh-of-trees. The resulting $AT^2$ bound of N $\log^2$ N( $|G|$ + $\log^2$ N) is within a few log $N$ factors of the lower bound of $AT^2$ = $\Omega$( N $|G|$ log N) shown in Section 5 even after accounting for the word model vs. bit model distinction. In Section 4 we showed that in a wide variety of cases, including the ones above, the time bounds can can be achieved even when $|G|$ is not known beforehand.

Several variations on the Beta Function Problem are possible. As described in Section 1, at the end of the Beta Operation, either the $|G|$ nontrivial (i, $y_i$) pairs end up in the first $|G|$ processors or else each $y_i$ appears at processor $i$. In some applications it may be appropriate to end the computation with every processor holding the reduction, $y_g$, corresponding to the group of its original (g, v)-pair. This problem is like computing $|G|$ independent census functions [LV81] in parallel. Let us call it the Beta/Census Problem. This can be implemented by first computing the $y_g$'s and then running the Beta Operation in reverse. One can run the Beta Operation in reverse if a trace of the forward Beta Operation was stored in the network. In general, this may be costly in terms of memory (=area). Fortunately, the algorithms demonstrated in this paper can be augmented to solve the Beta/Census Problem with only a constant factor increase in time and area.

It is interesting to note that the Beta Operation can be done probabilistically on the hypercube in time O(log N). Remember that the $O(\log^2 |G|)$ term comes solely from the sorting subroutine in Phase 1. If we use Flashsort [RV83] to sort, the bound obviously follows.

## Acknowledgements

The authors would like to thank Jeffrey D. Ullman for many constructive discussions and comments.

## References

[BH82] Borodin, A. and Hopcroft, J., "Routing, Merging and Sorting on Parallel Models of Computation", ***Proc. 14th ACM STOC***, **1982.**

[CS85] Cole, R. and Siegel, A., "Optimal VLSI circuits for Sorting", Technical Report 172, Computer Science Department, NYU, September 1985.

[GMU] Goldschlager, L., Mayr, E. and Ullman, J., ***Parallel Computation,*** in preparation.

[Hi85] Hillis, D., ***The Connection Machine,*** MIT Press, 1985.

[Ho85] Hochschild, P., Ph.D. Thesis, Stanford University: "Resource-Efficient Parallel Algorithms," Tech. Report STAN-CS-85-1073, July 1985.

[LV81] Lipton, R. and Valdes, J. , "Census functions: an approach to VLSI upper bounds," ***Proc. 21st IEEE FOCS,*** 1981.

[NS82] Nassimi, D. and Sahni, S., "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," JACM, vol 29, ***No. 3,*** July 1982.

[RV83] Reif, J. and Valiant, L., "A Logarithmic Time Sort for Linear Size Networks," ***Proc. 15th ACM STOC,*** 1983.

[U84] Ullman, J., ***Computational Aspects of VLSI,*** Computer Science Press, 1984.