A PARALLEL BUTTERFLY ALGORITHM

JACK POULSON*, LAURENT DEMANET†, NICHOLAS MAXWELL‡, AND LEXING YING§

Abstract. The *butterfly algorithm* is a fast algorithm which approximately evaluates a discrete analogue of the integral transform $\int_{\mathbb{R}^d} K(x,y)g(y)dy$ at large numbers of target points when the kernel, $K(x,y)$, is approximately low-rank when restricted to subdomains satisfying a certain simple geometric condition. In d dimensions with $O(N^d)$ source and target points, when each appropriate submatrix of K is approximately rank-r, the running time of the algorithm is at most $O(r^2 N^d \log N)$. A parallelization of the butterfly algorithm is introduced which, assuming a message latency of α and per-process inverse bandwidth of β, executes in at most $O\left(r^2 \frac{N^d}{p} \log N + \left(\beta r \frac{N^d}{p} + \alpha\right) \log p\right)$ time using p processes. This parallel algorithm was then instantiated in the form of the open-source `DistButterfly` library for the special case where $K(x,y) = \exp(i\Phi(x,y))$, where $\Phi(x,y)$ is a black-box, sufficiently smooth, real-valued *phase function*. Experiments on Blue Gene/Q demonstrate impressive strong-scaling results for important classes of phase functions. Using quasi-uniform sources, hyperbolic Radon transforms and an analogue of a 3D generalized Radon transform were respectively observed to strong-scale from 1-node/16-cores up to 1024-nodes/16,384-cores with greater than 90% and 82% efficiency, respectively. These experiments at least partially support the theoretical argument that, given $p = O(N^d)$ processes, the running-time of the parallel algorithm is $O\left((r^2 + \beta r + \alpha) \log N\right)$.

Key words. butterfly algorithm, Egorov operator, Radon transform, parallel, Blue Gene/Q

AMS subject classifications. 65R10, 65Y05, 65Y20, 44A12

1. Introduction.

The *butterfly algorithm* [24, 25, 34, 7] provides an efficient means of (approximately) applying any integral operator

$$(\mathcal{K}g)(x) = \int_Y K(x,y)g(y)dy$$

whose *kernel*, $K : X \times Y \to \mathbb{C}$, satisfies the condition that, given any *source box*, $B \subset Y$, and *target box*, $A \subset X$, such that the product of their diameters is less than some fixed constant, the restriction of K to $A \times B$, henceforth denoted $K|_{A\times B}$, is approximately low-rank in a point-wise sense. More precisely, when K is restricted to any such subdomain $A \times B$, there exists a low-rank separable approximation

$$\left| K(x,y) - \sum_{t=0}^{r-1} u_t^{AB}(x) v_t^{AB}(y) \right| \leq \epsilon, \quad \forall\, x \in A,\, y \in B,$$

where the rank, r, depends at most polylogarithmically on $1/\epsilon$.

If the source function, $g(y)$, is lumped into some finite set of points, $I_Y \subset Y$, the low-rank separable representation implies an approximation

$$\left| f^{AB}(x) - \sum_{t=0}^{r-1} u_t^{AB}(x) \delta_t^{AB} \right| \leq \epsilon \, \|g_B\|_1, \quad \forall\, x \in A,$$

*Department of Mathematics, Stanford University, 450 Serra Mall, Stanford, CA 94305 (poulson@stanford.edu).

†Department of Mathematics, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139 (laurent@math.mit.edu)

‡Department of Mathematics, University of Houston, 651 PGH, Houston, TX 77004 (nmaxwell@math.uh.edu)

§Department of Mathematics and ICME, Stanford University, 450 Serra Mall, Stanford, CA 94305 (lexing@math.stanford.edu).

where

$$f^{AB}(x) \equiv \sum_{y \in I_Y \cap B} K(x,y) g(y), \quad x \in A, \tag{1.1}$$

represents the potential generated in box A due to the sources in box B, $\|g_B\|_1$ is the ℓ_1 norm of g over its support in B, and each *expansion weight*, δ_t^{AB}, could simply be chosen as

$$\delta_t^{AB} = \sum_{y \in I_Y \cap B} v_t^{AB}(y) g(y).$$

The first step of the butterfly algorithm is to partition the source domain, Y, into a collection of boxes which are sufficiently small such that the product of each of their diameters with that of the entire target domain, X, satisfies the kernel's low-rank criterion. Then, for each box B in this initial partitioning of the source domain, the expansion weights, $\{\delta_t^{XB}\}_{t=0}^{r-1}$, for approximating the potential over X generated by the sources in B, f^{XB}, can be cheaply computed as $\delta_t^{XB} := \sum_{y \in I_Y \cap B} v_t^{XB}(y) g(y)$. In d dimensions, if there are N^d such source boxes, each containing $O(1)$ sources, then this initialization step only requires $O(rN^d)$ work.

Upon completion of the butterfly algorithm, we will have access to a much more useful set of expansion weights, those of $\{f^{AY}\}_A$, where each box A is a member of a sufficiently fine partitioning of the target domain such that the product of its diameter with that of the entire source domain satisfies the low-rank condition. Then, given any $x \in X$, there exists a box $A \ni x$ within which we may cheaply evaluate the approximate solution

$$f(x) = f^{AY}(x) \approx \sum_{t=0}^{r-1} u_t^{AY}(x) \delta_t^{AY}.$$

If we are interested in evaluating the solution at N^d target points, then the final evaluation phase clearly requires $O(rN^d)$ work.

The vast majority of the work of the butterfly algorithm lies in the translation of the expansion weights used to approximate the initial potentials, $\{f^{XB}\}_B$, into those which approximate the final potentials, $\{f^{AY}\}_A$. This transition is accomplished in $\log_2 N$ stages, each of which expends at most $O(r^2 N^d)$ work in order to map the expansion weights for the N^d interactions between members of a target domain partition, P_X, and a source domain partition, P_Y, into weights which approximate the N^d interactions between members of a *refined* target domain partition, P'_X, and a *coarsened* source domain partition, P'_Y. In particular, N is typically chosen to be a power of two such that, after ℓ stages, each dimension of the target domain is partitioned into 2^ℓ equal-sized intervals, and each dimension of the source domain is like-wise partitioned into $N/2^\ell$ intervals. This process is depicted for a simple one-dimensional problem, with $N = 8$, in Fig. 1.1, and, from now on, we will use the notation $\mathcal{T}_X(\ell)$ and $\mathcal{T}_Y(\ell)$ to refer to the sets of subdomains produced by the partitions for stage ℓ, where the symbol $\mathcal{T}$ hints at the fact that these are actually trees. Note that the root of $\mathcal{T}_X$, $\{X\}$, is at stage 0, whereas the root of $\mathcal{T}_Y$, $\{Y\}$, is at stage $\log_2 N$.

1.1. Merging and splitting. A cursory inspection of Fig. 1.1 reveals that each step of a 1D butterfly algorithm consists of many instances of transforming two potentials supported over neighboring source boxes and the same target box, into two

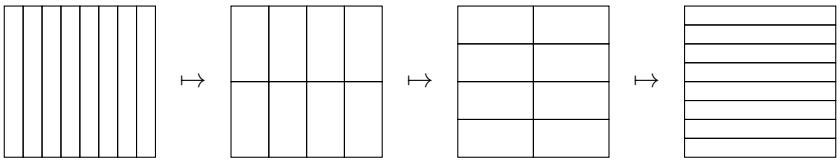

FIG. 1.1. *The successive partitions of the product space $X \times Y$ during a 1D butterfly algorithm with $N = 8$. This figure has an additional matrix-centric interpretation: each rectangle is an approximately low-rank submatrix of the discrete 1D integral operator.*

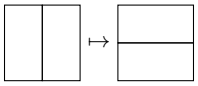

FIG. 1.2. *The fundamental operation in the butterfly algorithm: translating from two potentials with neighboring source domains and equal target domains into the corresponding potentials with neighboring target domains but the same combined source domain.*

potentials over the union of the two source boxes but only neighboring halves of the original target box (see Fig. 1.2). The generalization from one to d dimensions is obvious: 2^d neighboring source boxes are merged and the shared target box is split into 2^d subboxes.

Suppose that we are given a pair of target boxes, say A and B, and we define $\{B_j\}_j$ as the set of 2^d subboxes of B resulting from cutting each dimension of the box B into two equal halves. We shall soon see that, if each pair (A, B_j) satisfies the kernel's approximate low-rank criterion, then it is possible to compute a linear transformation which, to some predefined accuracy, maps any set of $2^d r$ weights representing potentials $\{f^{AB_j}\}_j$ into the $2^d r$ weights for the corresponding potentials after the merge-and-split procedure, say $\{f^{A_j B}\}_j$, where the 2^d subboxes $\{A_j\}_j$ of A are defined analogously to those of B.

In the case where the source and targets are quasi-uniformly distributed, $N^d/2^d$ such linear transformations need to be applied in each of the $\log_2 N$ stages of the algorithm, and so, if the corresponding matrices for these linear transformations have all been precomputed, the per-process cost of the butterfly algorithm is at most $O(r^2 N^d \log N)$ operations [25].

1.2. Equivalent sources. The original approach to the butterfly algorithm [24, 25] has an elegant physical interpretation and provides a straight-forward construction of the linear operators which translate weights from one stage to the next. The primary tool is a (strong) rank-revealing QR (RRQR) factorization [18, 9], which yields an accurate approximation of a numerically low-rank matrix in terms of linear combinations of a few of its columns. We will soon see how to manipulate an RRQR into an *interpolative decomposition* (ID) [25, 23],

$$\mathsf{K} \approx \hat{\mathsf{K}}\hat{\mathsf{Z}}, \tag{1.2}$$

where $\hat{\mathsf{K}}$ is a submatrix of $r \ll \min(m, n)$ columns of the $m \times n$ matrix K. While randomized algorithms for building IDs may often be more efficient [33], we will discuss deterministic RRQR-based constructions for the sake of simplicity.

Suppose that we have already computed a (truncated) RRQR decomposition

$$\mathsf{K}\Pi \approx \mathsf{Q}\begin{pmatrix} R_L & R_R \end{pmatrix}, \tag{1.3}$$

where $\mathsf{\Pi}$ is a permutation matrix, Q consists of r mutually orthonormal columns, and $\mathsf{R_L}$ is an invertible $r \times r$ upper-triangular matrix (otherwise a smaller RRQR could be formed). Then $\mathsf{K\Pi} \approx \hat{\mathsf{K}} \begin{pmatrix} I & R_L^{-1} R_R \end{pmatrix}$, where $\hat{\mathsf{K}} \equiv \mathsf{QR_L}$ is the subset of columns of K selected during the pivoted QR factorization (the first r columns of $\mathsf{K\Pi}$), and

$$\hat{\mathsf{Z}} \equiv \begin{pmatrix} I & R_L^{-1} R_R \end{pmatrix} \mathsf{\Pi^H} \tag{1.4}$$

is an $r \times n$ interpolation matrix such that $\mathsf{K} \approx \hat{\mathsf{K}}\hat{\mathsf{Z}}$, which completes our interpolative decomposition of K. We note that, although $\hat{\mathsf{Z}}$ is only guaranteed to be computed stably from a *strong* RRQR factorization, Businger-Golub pivoting [5] is typically used in practice in combination with applying the numerical pseudoinverse of R_L rather than R_L^{-1} [23].

The algorithms of [24, 25] exploit the fact that the interpolation matrix, $\hat{\mathsf{Z}}$, can be applied to a dense vector of n "sources" in order to produce an approximately equivalent set of r sources, in the sense that, for any vector $\mathsf{g} \in \mathbb{C}^n$,

$$\mathsf{Kg} \approx \hat{\mathsf{K}}(\hat{\mathsf{Z}}\mathsf{g}) = \hat{\mathsf{K}}\hat{\mathsf{g}}. \tag{1.5}$$

Or, equivalently, $\mathsf{Kg} \approx \mathsf{K}\hat{\mathsf{g}}_\mathsf{E}$, where $\hat{\mathsf{g}}_\mathsf{E}$ is the appropriate extension by zero of $\hat{\mathsf{g}} \in \mathbb{C}^r$ into $\mathbb{C}^n$. $\hat{\mathsf{Z}}$ therefore provides a fast mechanism for producing an approximately equivalent *sparse* source vector, $\hat{\mathsf{g}}_\mathsf{E}$, given any (potentially dense) source vector, g.

These sparse (approximately) equivalent sources can then be used as the expansion weights resulting from the low-rank approximation

$$\left| \mathsf{K}(\mathsf{i},\mathsf{j}) - \sum_{t=0}^{r-1} \mathsf{K}(\mathsf{i},\mathsf{j_t})\hat{\mathsf{z}}_\mathsf{t}(\mathsf{j}) \right| \leq \epsilon\, s(r,n), \quad \forall\, i,j,$$

namely,

$$\left\| \mathsf{Kg} - \hat{\mathsf{K}}\hat{\mathsf{g}} \right\|_\infty \leq \epsilon\, s(r,n) \|\mathsf{g}\|_1,$$

where $\mathsf{K}(:,\mathsf{j_t})$ is the t'th column of $\hat{\mathsf{K}}$, $\hat{\mathsf{z}}_\mathsf{t}$ is the t'th row of $\hat{\mathsf{Z}}$, $\hat{\mathsf{g}} \equiv \hat{\mathsf{Z}}\mathsf{g}$, and $s(r,n)$, which is bounded by a low-degree polynomial in r and n, is an artifact of RRQR factorizations yielding suboptimal low-rank decompositions [18].

If the matrix K and vector g were constructed such that $\mathsf{K}(\mathsf{i},\mathsf{j}) = K(x_i, y_j)$ and $\mathsf{g}(j) = g(y_j)$, for some set of source points $\{y_j\}_j \subset B$ and target points $\{x_i\}_i \subset A$, then the previous equation becomes

$$\left| f^{AB}(x_i) - \sum_{t=0}^{r-1} K(x_i, y_{j_t})\hat{g}(y_{j_t}) \right| \leq \epsilon\, s(r,n) \|g_B\|_1, \quad \forall\, i,$$

where we have emphasized the interpretation of the t'th entry of the equivalent source vector $\hat{\mathsf{g}}$ as a discrete source located at the point y_{j_t}, i.e., $\hat{g}(y_{j_t})$. We will now review how repeated applications of IDs can yield interpolation matrices which take $2^d r$ sources from 2^d neighboring source boxes and produce (approximately) equivalent sets of r sources valid over smaller target domains.

1.3. Translating equivalent sources. We will begin by considering the one-dimensional case, as it lends itself to a matrix-centric discussion: Let B_0 and B_1 be two neighboring source intervals of the same size, let A be a target interval of

sufficiently small width, and let $\hat{\mathsf{K}}_{\mathsf{AB}_0}$ and $\hat{\mathsf{K}}_{\mathsf{AB}_1}$ be subsets of columns from $\mathsf{K}|_{\mathsf{AB}_0}$ and $\mathsf{K}|_{\mathsf{AB}_1}$ generated from their interpolative decompositions with interpolation matrices $\hat{\mathsf{Z}}_{\mathsf{AB}_0}$ and $\hat{\mathsf{Z}}_{\mathsf{AB}_1}$. Then, for any source vector g, the vector of potentials over target box A generated by the sources in box B_n can be cheaply approximated as

$$\mathsf{f}_{\mathsf{AB}_n} \approx \hat{\mathsf{K}}_{\mathsf{AB}_n}\, \hat{\mathsf{g}}_{\mathsf{AB}_n} \left(= \hat{\mathsf{K}}_{\mathsf{AB}_n}\, \hat{\mathsf{Z}}_{\mathsf{AB}_n}\, \mathsf{g}|_{\mathsf{B}_n} \right),\ n = 0, 1.$$

If we then define $B = B_0 \cup B_1$ and halve the interval A into A_0 and A_1, then the products of the widths of each A_m with the entire box B is equal to that of A with each B_n, and so, due to the main assumption of the butterfly algorithm, $\mathsf{K}|_{\mathsf{A}_m\mathsf{B}}$ must also be numerically low-rank.

If we then split each $\hat{\mathsf{K}}_{\mathsf{AB}_n}$ into the two submatrices

$$\hat{\mathsf{K}}_{\mathsf{AB}_n} \rightarrow \begin{bmatrix} \hat{\mathsf{K}}_{\mathsf{A}_0\mathsf{B}_n} \\ \hat{\mathsf{K}}_{\mathsf{A}_1\mathsf{B}_n} \end{bmatrix},$$

we can write

$$\begin{pmatrix} \mathsf{f}_{\mathsf{A}_0\mathsf{B}} \\ \mathsf{f}_{\mathsf{A}_1\mathsf{B}} \end{pmatrix} \approx \begin{bmatrix} \hat{\mathsf{K}}_{\mathsf{A}_0\mathsf{B}_0} & \hat{\mathsf{K}}_{\mathsf{A}_0\mathsf{B}_1} \\ \hat{\mathsf{K}}_{\mathsf{A}_1\mathsf{B}_0} & \hat{\mathsf{K}}_{\mathsf{A}_1\mathsf{B}_1} \end{bmatrix} \begin{pmatrix} \hat{\mathsf{g}}_{\mathsf{AB}_0} \\ \hat{\mathsf{g}}_{\mathsf{AB}_1} \end{pmatrix}$$

and recognize that the two submatrices $\left[\hat{\mathsf{K}}_{\mathsf{A}_m\mathsf{B}_0}, \hat{\mathsf{K}}_{\mathsf{A}_m\mathsf{B}_1}\right]$, $m = 0, 1$, consist of subsets of columns of $\mathsf{K}|_{\mathsf{A}_m\mathsf{B}}$, which implies that they must also have low-rank interpolative decompositions, say $\hat{\mathsf{K}}_{\mathsf{A}_m\mathsf{B}}\hat{\mathsf{Z}}_{\mathsf{A}_m\mathsf{B}}$. Thus,

$$\begin{pmatrix} \mathsf{f}_{\mathsf{A}_0\mathsf{B}} \\ \mathsf{f}_{\mathsf{A}_1\mathsf{B}} \end{pmatrix} \approx \begin{bmatrix} \hat{\mathsf{K}}_{\mathsf{A}_0\mathsf{B}}\hat{\mathsf{Z}}_{\mathsf{A}_0\mathsf{B}} \\ \hat{\mathsf{K}}_{\mathsf{A}_1\mathsf{B}}\hat{\mathsf{Z}}_{\mathsf{A}_1\mathsf{B}} \end{bmatrix} \begin{pmatrix} \hat{\mathsf{g}}_{\mathsf{AB}_0} \\ \hat{\mathsf{g}}_{\mathsf{AB}_1} \end{pmatrix} = \begin{pmatrix} \hat{\mathsf{K}}_{\mathsf{A}_0\mathsf{B}}\hat{\mathsf{g}}_{\mathsf{A}_0\mathsf{B}} \\ \hat{\mathsf{K}}_{\mathsf{A}_1\mathsf{B}}\hat{\mathsf{g}}_{\mathsf{A}_1\mathsf{B}} \end{pmatrix},$$

where we have defined the new equivalent sources, $\hat{\mathsf{g}}_{\mathsf{A}_m\mathsf{B}}$, as

$$\hat{\mathsf{g}}_{\mathsf{A}_m\mathsf{B}} \equiv \hat{\mathsf{Z}}_{\mathsf{A}_m\mathsf{B}} \begin{pmatrix} \hat{\mathsf{g}}_{\mathsf{AB}_0} \\ \hat{\mathsf{g}}_{\mathsf{AB}_1} \end{pmatrix}.$$

The generalization to d-dimensions should again be clear: 2^d sets of IDs should be stacked together, partitioned, and recompressed in order to form 2^d interpolation matrices of size $r \times 2^d r$, say $\{\hat{\mathsf{Z}}_{\mathsf{A}_m\mathsf{B}}\}_m$. These interpolation matrices can then be used to translate 2^d sets of equivalent sources from one stage to the next with at most $O(r^2)$ work. Recall that each of the $\log_2 N$ stage of the butterfly algorithm requires $O(N^d)$ such translations, and so, *if all necessary IDs have been precomputed*, the equivalent source approach yields an $O(r^2 N^d \log N)$ butterfly algorithm. There is, of course, an analogous approach based on row-space interpolation, which can be interpreted as constructing a small set of representative potentials which may then be cheaply interpolated to evaluate the potential over the entire target box.

Yet another approach would be to replace row/column-space interpolation with the low-rank approximation implied by a Singular Value Decomposition (SVD) and to construct the $2^d r \times 2^d r$ linear weight transformation matrices based upon the low-rank approximations used at successive levels. Since the low-rank approximations produced by SVDs are generally much tighter than those of rank-revealing factorizations, such an approach would potentially allow for lower-rank approximations to result in the same overall accuracy.

From now on, we will use the high-level notation that $\mathsf{T}^{\mathsf{A}\,\mathsf{B}_n}_{\mathsf{A}_m\mathsf{B}}$ is the *translation operator* which maps a weight vector from a low-rank decomposition over $A \times B_n$, $\mathsf{w}_{\mathsf{AB}_n}$, into that of a low-rank decomposition over $A_m \times B$, $\mathsf{w}_{\mathsf{A}_m\mathsf{B}}$. Clearly each merge and split operation involves a $2^d \times 2^d$ block matrix of such translation operators. Please see Algorithm 1.1 for a demonstration of the sequential algorithm from the point of view of translation operators, where the low-rank approximation of each block $\mathsf{K}|_{\mathsf{AB}}$ is denoted by $\mathsf{U}_{\mathsf{AB}}\mathsf{V}_{\mathsf{AB}}$, and we recall that the nodes of the trees active at the beginning of the ℓ'th stage of the algorithm are denoted by $\mathcal{T}_X(\ell)$ and $\mathcal{T}_Y(\ell)$.

Algorithm 1.1: Sequential butterfly algorithm over a d-dimensional domain with N^d source and target points.

$\mathcal{A} := \mathcal{T}_X(0),\ \mathcal{B} := \mathcal{T}_Y(0)$
$\left(\mathcal{A} = \{X\},\ \cup_{B\in\mathcal{B}} B = Y,\ \mathrm{card}(\mathcal{B}) = N^d\right)$
`// Initialize weights`
foreach $B \in \mathcal{B}$ **do**
  $\mathsf{w}_{\mathsf{XB}} := \mathsf{V}_{\mathsf{XB}}\,\mathsf{g}_{\mathsf{B}}$
`// Translate weights`
for $\ell = 0, ..., \log_2 N - 1$ **do**
  $\tilde{\mathcal{A}} := \mathrm{children}(\mathcal{A}),\ \tilde{\mathcal{B}} := \mathrm{parents}(\mathcal{B})$
  foreach $(\tilde{A}, \tilde{B}) \in \tilde{\mathcal{A}} \times \tilde{\mathcal{B}}$ **do**
    $\mathsf{w}_{\tilde{\mathsf{A}}\tilde{\mathsf{B}}} := \mathsf{0}$
  foreach $(A, B) \in \mathcal{A} \times \mathcal{B}$ **do**
    $\{A_c\}_{c=0}^{2^d-1} := \mathrm{children}(A),\ B_p := \mathrm{parent}(B)$
    foreach $c = 0, ..., 2^d - 1$ **do**
      $\mathsf{w}_{\mathsf{A}_c\mathsf{B}_p}$ += $\mathsf{T}^{\mathsf{AB}}_{\mathsf{A}_c\mathsf{B}_p}\mathsf{w}_{\mathsf{AB}}$
  $\mathcal{A} := \tilde{\mathcal{A}},\ \mathcal{B} := \tilde{\mathcal{B}}$
$\left(\mathrm{card}(\mathcal{A}) = N^d,\ \cup_{A\in\mathcal{A}} A = X,\ \mathcal{B} = \{Y\}\right)$
`// Final evaluations`
foreach $A \in \mathcal{A}$ **do**
  $\mathsf{f}_{\mathsf{AY}} := \mathsf{U}_{\mathsf{AY}}\,\mathsf{w}_{\mathsf{AY}}$

1.4. Avoiding quadratic precomputation. The obvious drawback to ID and SVD-based approaches is that the precomputation of the $O(N^d \log N)$ necessary low-rank approximations requires at least $O(N^{2d})$ work with any black-box algorithm, as the first stage of the butterfly algorithm involves $O(N^d)$ matrices of height N^d. If we assume additional features of the underlying kernel, we may accelerate these precomputations [31] or, in some cases, essentially avoid them altogether [7, 34].

We focus on the latter case, where the kernel is assumed to be of the form

$$K(x, y) = e^{i\Phi(x,y)}, \tag{1.6}$$

where $\Phi : X \times Y \to \mathbb{R}$ is a sufficiently smooth[1] *phase function*. Due to the assumed smoothness of Φ, it was shown in [7, 13, 21] that the precomputation of IDs can be replaced with analytical interpolation of the row or column space of the numerically

[1] Formally, (Q,R)-analytic [7, 13]

low-rank submatrices using tensor-product Chebyshev grids. In particular, in the first half of the algorithm, while the source boxes are small, interpolation is performed within the column space, and in the middle of the algorithm, when the target boxes become as small as the source boxes, the algorithm switches to analytical row-space interpolation.

An added benefit of the tensor-product interpolation is that, if a basis of dimension q is used in each direction, so that the rank of each approximation is $r = q^d$, the weight translation cost can be reduced from $O(r^2)$ to $O(q^{d+1}) = O(r^{1+1/d})$. However, we note that the cost of the *middle-switch* from column-space to row-space interpolation, in general, requires $O(r^2)$ work for each set of weights. But this cost can also be reduced to $O(r^{1+1/d})$ when the phase function, $\Phi(x, y)$, also has a tensor-product structure, e.g., $x \cdot y$. Lastly, performing analytical interpolation allows one to choose the precise locations of the target points *after* forming the final expansion weights, $\{\mathsf{w}_{\mathsf{AY}}\}_A$, and so the result is best viewed as a potential field which only requires $O(r)$ work to evaluate at any given point in the continuous target domain.

2. Parallelization. We now present a parallelization of butterfly algorithms which is high-level enough to handle both general-purpose [24, 25] and analytical [7] low-rank interpolation. We will proceed by first justifying our communication cost model, then step through the simplest parallel case, where each box is responsible for a single interaction at a time, and then demonstrate how the communication requirements change when less processes are used. We will not discuss the precomputation phase in detail, as the factorizations computed at each level need to be exchanged between processes in the same manner as the weights, but the factorizations themselves are relatively much more expensive and can each be run sequentially (unless $p > N^d$). Readers interested in efficient parallel IDs should consult the communication-avoiding RRQR factorization of [14].

2.1. Communication cost model. All of our analysis makes use of a commonly-used [16, 30, 8, 3, 14] communication cost model that is as useful as it is simple: each process is assumed to only be able to simultaneously send and receive a single message at a time, and, when the message consists of n units of data (e.g., double-precision floating-point numbers), the time to transmit such a message between any two processes is $\alpha + \beta n$ [19, 2]. The α term represents the time required to send an arbitrarily small message and is commonly referred to as the message *latency*, whereas $1/\beta$ represents the number of units of data which can be transmitted per unit of time once the message has been initiated.

There also exist more sophisticated communication models, such as *LogP* [11] and its extension, *LogGP* [1], but the essential differences are that the former separates the local software overhead (the 'o' in 'LogP') from the network latency and the latter compensates for very large messages potentially having a different transmission mechanism. An arguably more important detail left out of the $\alpha + \beta n$ model is the issue of *network conflicts* [30, 8], that is, when multiple messages compete for the bandwidth available on a single communication link. We will ignore network conflicts since they greatly complicate our analysis and require the specialization of our cost model to a particular network topology.

2.2. High-level approach with N^d processes. The proposed parallelization is easiest to discuss for cases where the number of processes is equal to N^d, the number of pairwise source and target box interactions represented at each level of the butterfly algorithm. Recall that each of these interactions is represented with r

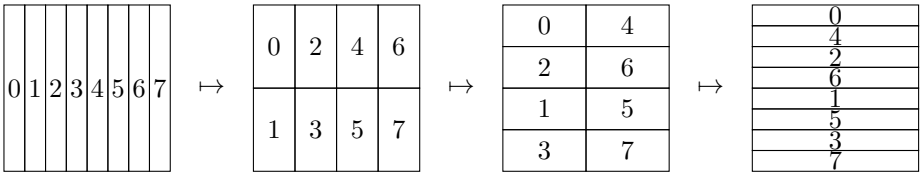

FIG. 2.1. *The data distribution throughout the execution of a 1D butterfly algorithm with $p = N = 8$. Notice that the distribution of the target domain upon output is the same as the input distribution of the source domain, but with the binary process ranks reversed.*

expansion weights, where r is a small number which should depend polylogarithmically on the desired accuracy. We will now present a scheme which assigns each process one set of expansion weights at each stage and has a runtime of $O((r^2 + \beta r + \alpha) \log N)$.

Consider the data distribution scheme shown in Fig. 2.1 for a 1D butterfly algorithm with both the problem size, N, and number of processes, p, set to eight. In the beginning of the algorithm (the left-most panel), each process is assigned one source box and need only expend $O(r)$ work in order to initialize its weights, and, at the end of the algorithm (the right-most panel), each process can perform $O(r)$ flops in order to evaluate the potential over its target box. Since each stage of the butterfly algorithm involves linearly transforming 2^d sets of weights from one level to the next, e.g., via

$$\begin{pmatrix} \mathsf{w}_{\mathsf{A}_0\mathsf{B}} \\ \mathsf{w}_{\mathsf{A}_1\mathsf{B}} \end{pmatrix} = \begin{bmatrix} \mathsf{T}^{\mathsf{A}\,\mathsf{B}_0}_{\mathsf{A}_0\mathsf{B}} & \mathsf{T}^{\mathsf{A}\,\mathsf{B}_1}_{\mathsf{A}_0\mathsf{B}} \\ \mathsf{T}^{\mathsf{A}\,\mathsf{B}_0}_{\mathsf{A}_1\mathsf{B}} & \mathsf{T}^{\mathsf{A}\,\mathsf{B}_1}_{\mathsf{A}_1\mathsf{B}} \end{bmatrix} \begin{pmatrix} \mathsf{w}_{\mathsf{A}\mathsf{B}_0} \\ \mathsf{w}_{\mathsf{A}\mathsf{B}_1} \end{pmatrix},$$

pairs of processes will need to coordinate in order to perform parallel matrix-vector multiplications of size $2r \times 2r$. Because each process initially owns only half of the vector that must be linearly transformed, it is natural to locally compute half of the linear transformation, for example, $\mathsf{T}^{\mathsf{A}\,\mathsf{B}_0}_{\mathsf{A}_0\mathsf{B}}\,\mathsf{w}_{\mathsf{A}\mathsf{B}_0}$ and $\mathsf{T}^{\mathsf{A}\mathsf{B}_0}_{\mathsf{A}_1\mathsf{B}}\,\mathsf{w}_{\mathsf{A}\mathsf{B}_0}$, and to combine the results with those computed by the partner process. Furthermore, the output weights of such a linear transformation should also be distributed, and so only r entries of data need be exchanged between the two processes, for a cost of $\alpha + \beta r$. Since the cost of the local transformation is at most $O(r^2)$, and only $O(r)$ work is required to sum the received data, the cost of the each stage is at most $O(r^2 + \beta r + \alpha)$.

In the first, second, and third stages of the algorithm, process 0 would respectively pair with processes 1, 2, and 4, and its combined communication and computation cost would be $O((r^2+\beta r+\alpha) \log N)$. Every other process need only perform the same amount of work and communication, and it can be seen from the right-most panel of Fig. 2.1 that, upon completion, each process will hold an approximation for the potential generated over a single target box resulting from the entire set of sources. In fact, the final distribution of the target domain can be seen to be the same as the initial distribution of the source domain, but with the bits of the owning processes reversed (see Fig. 2.2).

The bit-reversal viewpoint is especially useful when discussing the parallel algorithm in higher dimensions, as more bookkeeping is required in order to precisely describe distribution of the product space of two multi-dimensional domains. It can be seen from the one-dimensional case shown in Fig. 2.2 that the bitwise rank-reversal is the result of each stage of the parallel algorithm moving the finest-scale bitwise partition of the source domain onto the target domain. In order to best visualize the

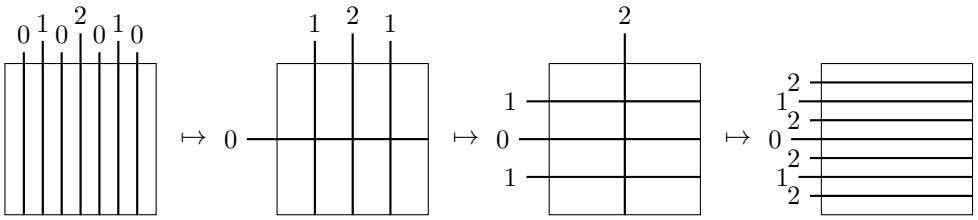

FIG. 2.2. *An alternative view of the data distributions used during a 1D parallel butterfly algorithm with $p = N = 8$ based upon bitwise bisections: processes with their j'th bit set to zero are assigned to the left or upper side of the partition, while processes with the j'th bit of their rank set to one are assigned to the other side.*

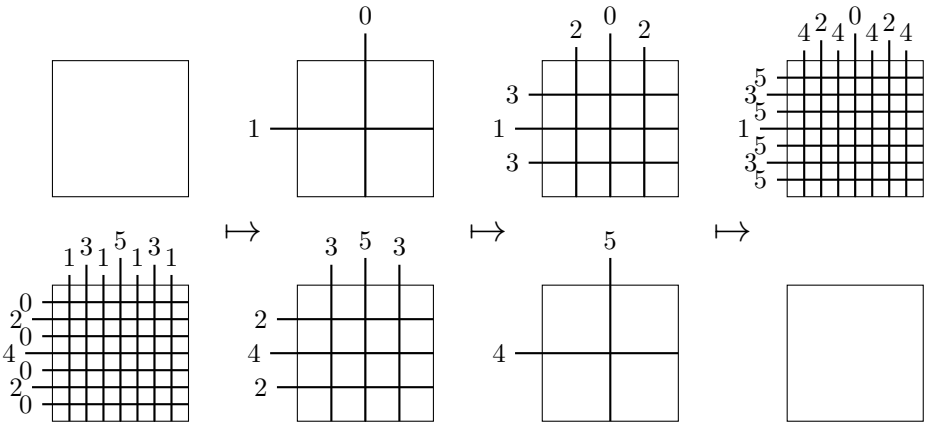

FIG. 2.3. *The data distributions of the source domain (bottom) and target domain (top) throughout the execution of a 2D butterfly algorithm with $p = N^2 = 64$ expressed using bitwise process rank partitions. Notice that the product space remains evenly distributed throughout the entire computation.*

structure of the equivalent process in higher dimensions, it is useful to switch from the matrix-centric viewpoint used in the 1D example of Fig. 2.2 to the dyadic viewpoint of the two-dimensional example of Fig. 2.3.

The main generalization required for the multi-dimensional algorithm is that, rather than pairs of processes interacting, 2^d processes will need to work together in order to map 2^d sets of weights from one level to the next. Just as in the one-dimensional case, each process need only receive one of the 2^d sets of weights of the result, and the appropriate generalization of each pairwise exchange is a call to `MPI_Reduce_scatter_block` over a team of 2^d processes, which only requires each process to send d messages and $(2^d - 1)r$ entries of data, and to perform $(2^d - 1)r$ flops [30]. This communication pattern is precisely the mechanism in which the bitwise partitions are moved from the source domain to the target domain: teams of processes owning weights which neighbor in the source domain cooperate in order to produce the weights needed for the next level, which neighbor in the target domain.

It is thus easy to see that the per-process cost of the multi-dimensional parallel butterfly algorithm is also at most $O((r^2 + \beta r + \alpha) \log N)$. When analytical tensor-product interpolation is used, the local computation required for linearly mapping 2^d sets of weights from one stage to the next can be as low as $O(r^{1+1/d})$, which clearly

lowers the parallel complexity to $O((r^{1+1/d} + \beta r + \alpha) \log N)$. However, there is an additional $O(r^2)$ cost for transitioning from column-space to row-space interpolation in the middle of the algorithm when the phase function does not have tensor-product structure [7].

2.3. Algorithm for $p = N^d$ processes. In order to give precise pseudocode, it is useful to codify the bitwise bisections applied to X and Y in terms of two *stacks*, $\mathcal{D}_X$ and $\mathcal{D}_Y$. Each item in the stack is then uniquely specified by the dimension of the domain it is being applied to and the bit of the process rank used to determine which half of the domain it will be moved to. For instance, to initialize the bisection stack $\mathcal{D}_Y$ for the two-dimensional $p = 64$ case shown in Fig. 2.3, we might run the following steps:

Algorithm 2.1: Initialize bisection stacks in d-dimensions with p processes

$\mathcal{D}_X := \mathcal{D}_Y := \varnothing$
for $j = 0, ..., \log_2 p - 1$ **do**
  $\mathcal{D}_Y$.push($(j \bmod d, (\log_2 p - 1) - j)$)

This algorithm effectively cycles through the d dimensions bisecting based upon each of the $\log_2 p$ bits of the process ranks, starting with the most-significant bit (with index $\log_2 p - 1$).

Once the bisection stacks have been initialized, the data distribution at level ℓ is defined by sequentially *popping* the $d\ell$ bisections off the top of the $\mathcal{D}_Y$ stack and *pushing* them onto the $\mathcal{D}_X$ stack, which we may express as running $\mathcal{D}_X$.push($\mathcal{D}_Y$.pop()) $d\ell$ times. We also make use of the notation $\mathcal{D}_X(q)$ and $\mathcal{D}_Y(q)$ for the portions of X and Y which $\mathcal{D}_X$ and $\mathcal{D}_Y$ respectively assign to process q. Lastly, given p processes, we define the *bit-masking operator*, $\mathcal{M}_a^b(q)$, as

$$\mathcal{M}_a^b(q) = \{n \in [0, p) : \text{bit}_j(n) = \text{bit}_j(q),\ \forall j \notin [a, b)\}, \tag{2.1}$$

so that, as long as $0 \leq a \leq b < \log_2 p$, the cardinality of $\mathcal{M}_a^b(q)$, denoted $\text{card}(\mathcal{M}_a^b(q))$, is 2^{b-a}. In fact, it can be seen that stage ℓ of the $p = N^d$ parallel butterfly algorithm, listed as Algorithm 2.2, requires process q to perform a reduce-scatter summation over the team of 2^d processes denoted by $\mathcal{M}_{d\ell}^{d(\ell+1)}(q)$.

2.4. Algorithm for $p \leq N^d$ processes. We will now generalize the algorithm of the previous subsection to any case where a power-of-two number of processes less than or equal to N^d is used and show that the cost is at most

$$O\left(r^2 \frac{N^d}{p} \log N + \left(\beta r \frac{N^d}{p} + \alpha\right) \log p\right),$$

where N^d/p is the number of interactions assigned to each process during each stage of the algorithm. This cost clearly reduces to that of the previous subsection when $p = N^d$ and to the $O(r^2 N^d \log N)$ cost of the sequential algorithm when $p = 1$.

Consider the case where $p < N^d$, such as the 2D example in Fig. 2.4, where $p = 16$ and $N^2 = 64$. As shown in the bottom-left portion of the figure, each process is initially assigned a contiguous region of the source domain which contains multiple source boxes. If p is significantly less than N^d, it can be seen that the first several stages of the butterfly algorithm will not require any communication, as the linear transformations which coarsen the source domain (and refine the target domain) will take place over sets of source boxes assigned to a single process. More specifically,

Algorithm 2.2: Parallel butterfly algorithm with $p = N^d$ process from the point of view of process q.

```
// Initialize bitwise-bisection stacks
```
$\mathcal{D}_X := \mathcal{D}_Y := \varnothing$
for $j = 0, ..., \log_2 p - 1$ **do**
  $\mathcal{D}_Y$.push($((j \bmod d, (\log_2 p - 1) - j)$)
$(\mathcal{D}_X(q) = X,\ \cup_q \mathcal{D}_Y(q) = Y)$
$A := \mathcal{D}_X(q),\ B := \mathcal{D}_Y(q)$
```
// Initialize local weights
```
$\mathsf{w}_{\mathsf{XB}} := \mathsf{V}_{\mathsf{XB}}\, \mathsf{g}_{\mathsf{B}}$
```
// Translate the weights in parallel
```
for $\ell = 0, ..., \log_2(N) - 1$ **do**
  $\{A_c\}_{c=0}^{2^d-1} := \text{children}(A),\ \ B_p := \text{parent}(B)$
  foreach $c = 0, ..., 2^d - 1$ **do**
    $\mathsf{w}_{\mathsf{A_cB_p}} := \mathsf{T}^{\mathsf{AB}}_{\mathsf{A_cB_p}} \mathsf{w}_{\mathsf{AB}}$
  for $j = 0, ..., d - 1$ **do**
    $\mathcal{D}_X$.push($\mathcal{D}_Y$.pop())
  $A := \mathcal{D}_X(q),\ B := \mathcal{D}_Y(q)$
  $\mathsf{w}_{\mathsf{AB}} := \text{SumScatter}(\{\mathsf{w}_{\mathsf{A_cB_p}}\}_{c=0}^{2^d-1}, \mathcal{M}_{d\ell}^{d(\ell+1)}(q))$
$(\cup_q \mathcal{D}_X(q) = X,\ \mathcal{D}_Y(q) = Y)$
```
// Final evaluation
```
$\mathsf{f}_{\mathsf{AY}} := \mathsf{U}_{\mathsf{AY}}\, \mathsf{w}_{\mathsf{AY}}$

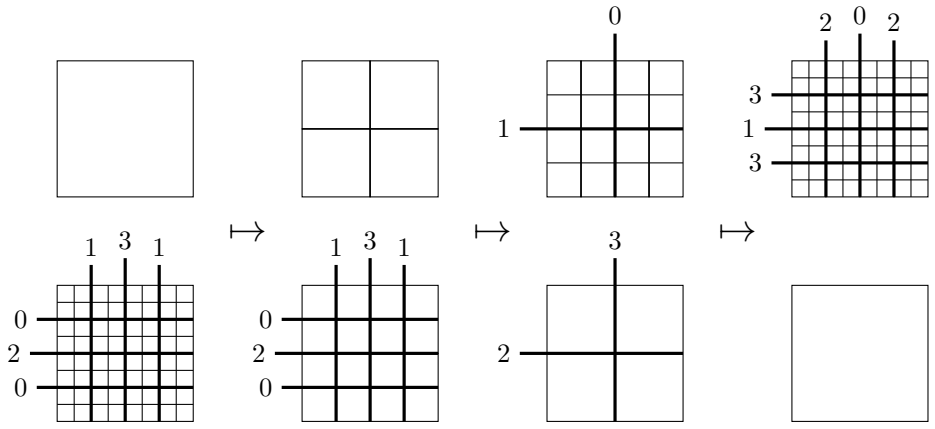

FIG. 2.4. *A 2D parallel butterfly algorithm with $N^d = 64$ and $p = 16$. The first stage does not require any communication, and the remaining $\lceil \log_{2^d} p \rceil = 2$ stages each require teams of 2^d processes to coordinate in order to perform $N^d/p = 4$ simultaneous linear transformations.*

when N is a power of two, we may decompose the number of stages of the butterfly algorithm, $\log_2 N$, as

$$\log_2 N = \lfloor \log_{2^d} \frac{N^d}{p} \rfloor + \lceil \log_{2^d} p \rceil,$$

where the former term represents the number of stages at the beginning of the butterfly algorithm which may be executed entirely locally, and the latter represents the number of stages requiring communication. In the case of Fig. 2.4, $\log_{2^d}(N^d/p) = 1$, and only the first stage does not require communication.

It is important to notice that each of the last $\lceil \log_{2^d} p \rceil$ stages still requires at most 2^d processes to interact, and so the only difference in the communication cost is that $O(rN^d/p)$ entries of data need to be exchanged within the team rather than just $O(r)$. Note that any power of two number of processes can be used with such an approach, though, when p is not an integer power of 2^d, the first communication step will involve between 2 and 2^{d-1} processes, while the remaining stages will involve teams of 2^d processes. Despite these corner cases, the total communication cost can easily be seen to be $O((\beta r(N^d/p) + \alpha) \log p)$, and the total computation cost is at most $O(r^2(N^d/p) \log N)$.

Algorithm 2.3 gives a precise prescription of these ideas using the bisection stacks and bit-masking operators defined in the previous subsection. Because each process can now have multiple source and target boxes assigned to it at each stage of the algorithm, we denote these sets of boxes as

$$\mathcal{T}_X(\ell)|_{\mathcal{D}_X(q)} = \{A \in \mathcal{T}_X(\ell) : A \in \mathcal{D}_X(q)\}, \text{ and}$$
$$\mathcal{T}_Y(\ell)|_{\mathcal{D}_Y(q)} = \{A \in \mathcal{T}_Y(\ell) : A \in \mathcal{D}_Y(q)\}.$$

3. Experimental results. As previously mentioned, our performance experiments focus on the class of integral operators whose kernels are of the form of Eq. (1.6). While this may seem overly restrictive, a large class of important transforms falls into this category, most notably: the Fourier transform, where $\Phi(x, y) = 2\pi x \cdot y$, back-projection [13], hyperbolic Radon transforms [20], and Egorov operators, which then provide a means of efficiently applying Fourier Integral Operators [7]. Due to the extremely special (and equally delicate) structure of Fourier transforms, a number of highly-efficient parallel algorithms already exist for both uniform [16, 26, 12] and non-uniform [27] Fourier transforms, and so we will instead concentrate on more sophisticated kernels. We note that the communication cost of the parallel 1D FFT mentioned in [16] is essentially equivalent to that of our parallel 1D butterfly algorithm.

Algorithm 2.3 was instantiated in the new `DistButterfly` library using black-box, user-defined phase functions, and the low-rank approximations and translation operators introduced in [7]. The library was written using `C++11` in order to template the implementation over the dimension of the problem, and all inter-process communication was expressed via the Message Passing Interface (MPI). All tests were performed on the Argonne Leadership Computing Facility's Blue Gene/Q installation using a port of Clang and IBM's ESSL 5.1, and kernel evaluations were accelerated by batching them together and calling ESSL's MASS routines, `vsin`, `vcos`, and `vsincos`. All calculations were performed with (64-bit) double-precision arithmetic.

The current implementation is written purely with MPI and does not employ multi-threading, and because Blue Gene/Q's cores require multiple threads in order to saturate the machine's floating-point units [32], it was empirically found that launching four MPI processes for each of the 16 cores on each node resulted in the best performance. All strong-scaling tests were therefore conducted in the range of 1-node/16-cores/64-processes and 1024-nodes/16,384-cores/65,536-processes.

Algorithm 2.3: Parallel butterfly algorithm for $p \leq N^d$ processes from the view of process q. p is assumed to be a power of two.

```
// Initialize bitwise-bisection stacks
```
$\mathcal{D}_X := \mathcal{D}_Y := \varnothing$
for $j = 0, ..., \log_2 p - 1$ **do**
  $\mathcal{D}_Y$.push($((j \bmod d, (\log_2 p - 1) - j)$)
$\Big(\mathcal{D}_X(q) = X,\ \cup_q \mathcal{D}_Y(q) = Y,\ \mathrm{card}\big(\mathcal{T}_Y(0)|_{\mathcal{D}_Y(q)}\big) = \frac{N^d}{p}\Big)$
$\mathcal{A} := \mathcal{T}_X(0)|_{\mathcal{D}_X(q)},\ \mathcal{B} := \mathcal{T}_Y(0)|_{\mathcal{D}_Y(q)}$
```
// Initialize local weights
```
foreach $B \in \mathcal{B}$ **do**
  $\mathsf{w_{XB}} := \mathsf{V_{XB}}\,\mathsf{g_B}$
```
// Translate weights in parallel
```
$s := \log_2 \left(\frac{N^d}{p} \bmod 2^d\right)$
for $\ell = 0, ..., \log_2 N - 1$ **do**
  $\tilde{\mathcal{A}} := \mathrm{children}(\mathcal{A}),\ \tilde{\mathcal{B}} := \mathrm{parents}(\mathcal{B})$
  foreach $(\tilde{A}, \tilde{B}) \in \tilde{\mathcal{A}} \times \tilde{\mathcal{B}}$ **do**
    $\mathsf{w_{\tilde{A}\tilde{B}}} := \mathsf{0}$
  foreach $(A, B) \in \mathcal{A} \times \mathcal{B}$ **do**
    $\{A_c\}_{c=0}^{2^d-1} := \mathrm{children}(A),\ B_p := \mathrm{parent}(B)$
    foreach $c = 0, ..., 2^d - 1$ **do**
      $\mathsf{w_{A_cB_p}} \mathrel{+}= \mathsf{T^{AB}_{A_cB_p}}\,\mathsf{w_{AB}}$
  if $\ell < \lfloor \log_{2^d} \frac{N^d}{p} \rfloor$ **then**
    $\mathcal{A} := \tilde{\mathcal{A}},\ \mathcal{B} := \tilde{\mathcal{B}}$
  else if $\ell = \lfloor \log_{2^d} \frac{N^d}{p} \rfloor$ *and* $s \neq 0$ **then**
    for $j = 0, ..., s - 1$ **do**
      $\mathcal{D}_X$.push($\mathcal{D}_Y$.pop())
    $\mathcal{A} := \mathcal{T}_X(\ell + 1)|_{\mathcal{D}_X(q)},\ \mathcal{B} := \mathcal{T}_Y(\ell + 1)|_{\mathcal{D}_Y(q)}$
    $\{\mathsf{w_{AB}}\}_{A \in \mathcal{A}, B \in \mathcal{B}} := \mathrm{SumScatter}(\{\mathsf{w_{\tilde{A}\tilde{B}}}\}_{\tilde{A} \in \tilde{\mathcal{A}}, \tilde{B} \in \tilde{\mathcal{B}}}, \mathcal{M}_{d\ell}^{d\ell+s}(q))$
  else
    for $j = 0, ..., d - 1$ **do**
      $\mathcal{D}_X$.push($\mathcal{D}_Y$.pop())
    $\mathcal{A} := \mathcal{T}_X(\ell + 1)|_{\mathcal{D}_X(q)},\ \mathcal{B} := \mathcal{T}_Y(\ell + 1)|_{\mathcal{D}_Y(q)}$
    $\{\mathsf{w_{AB}}\}_{A \in \mathcal{A}, B \in \mathcal{B}} := \mathrm{SumScatter}(\{\mathsf{w_{\tilde{A}\tilde{B}}}\}_{\tilde{A} \in \tilde{\mathcal{A}}, \tilde{B} \in \tilde{\mathcal{B}}}, \mathcal{M}_{d(\ell-1)+s}^{d\ell+s}(q))$
$\Big(\mathrm{card}\big(\mathcal{T}_X(\log_2 N)|_{\mathcal{D}_X(q)}\big) = \frac{N^d}{p},\ \cup_q \mathcal{D}_X(q) = X,\ \mathcal{D}_Y(q) = Y\Big)$
```
// Final evaluations
```
foreach $A \in \mathcal{A}$ **do**
  $\mathsf{f_{AY}} := \mathsf{U_{AY}}\,\mathsf{w_{AY}}$

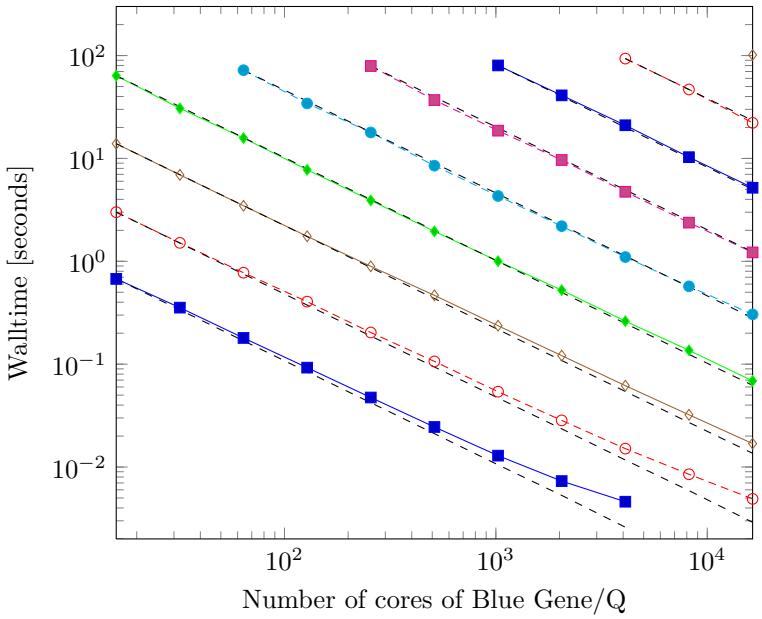

FIG. 3.1. *A combined strong-scaling and asymptotic complexity test for Algorithm 2.3 using analytical interpolation for a 2D hyperbolic Radon transform with numerical ranks of $r = 4^2$. From bottom-left to top-right, the tests involved N^2 source and target boxes with N equal to 128, 256, ..., 32768. Note that four MPI processes were launched per core in order to maximize performance.*

3.1. Hyperbolic Radon transforms. Our first set of experiments used a phased function of the form

$$\Phi((x_0, x_1), (h, p)) = 2\pi p\sqrt{x_0^2 + x_1^2 h^2},$$

which corresponds to a 2D (or 3D)[2] *hyperbolic Radon transform* [20] and has many applications in seismic imaging. For the strong-scaling tests, the numerical rank was fixed at $r = 4^2$, which corresponds to a tensor product of two four-point Chebyshev grids for each low-rank interaction.

As can be seen from Fig. 3.1, the performance of both the $N = 128$ and $N = 256$ problems continued to improve all the way to the $p = N^2$ limits (respectively, 16,384 and 65,536 processes). As expected, the larger problems display the best strong scaling, and the largest problem which would fit on one node, $N = 1024$, scaled from 64 to 65,536 processes with roughly 90.5% efficiency.

3.2. 3D Generalized Radon analogue. Our second test involves an analogue of a 3D generalized Radon transform [4] and again has applications in seismic imaging. As in [7], we use a phase function of the form

$$\Phi(x, p) = \pi\left(x \cdot p + \sqrt{\gamma(x,p)^2 + \kappa(x,p)^2}\right),$$

where

$$\gamma(x, p) = p_0\left(2 + \sin(2\pi x_0)\sin(2\pi x_1)\right)/3, \text{ and}$$
$$\kappa(x, p) = p_1\left(2 + \cos(2\pi x_0)\cos(2\pi x_1)\right)/3.$$

[2] If the degrees of freedom in the second and third dimensions are combined [20].

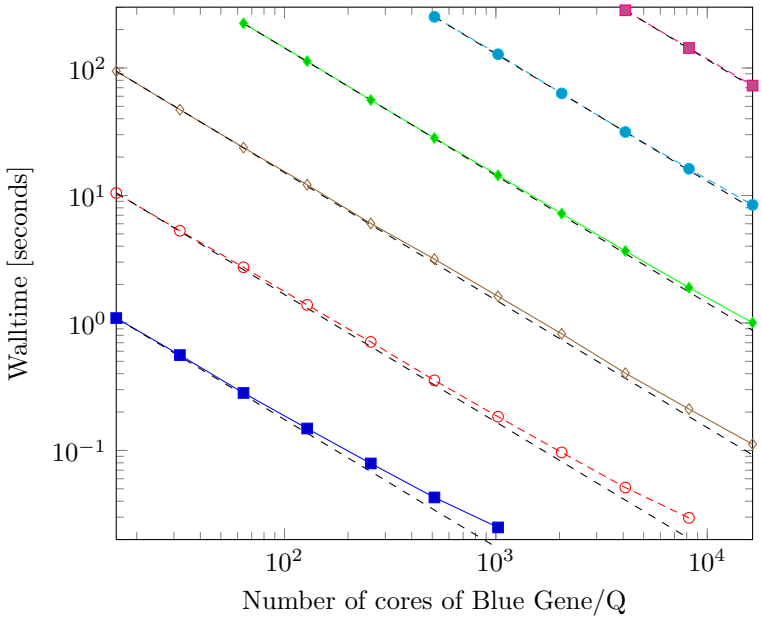

FIG. 3.2. *A combined strong-scaling and asymptotic complexity test for Algorithm 2.3 using analytical interpolation for a 3D generalized Radon analogue with numerical rank $r = 5^3$. From bottom-left to top-right, the tests involved N^3 source and target boxes with N equal to 16, 32, ..., 512. Note that four MPI processes were launched per core in order to maximize performance.*

Notice that, if not for the non-linear square-root term, the phase function would be equivalent to that of a Fourier transform. The numerical rank was chosen to be $r = 5^3$ in order to provide roughly one percent relative error in the supremum norm, and due to both the increased rank, increased dimension, and more expensive phase function, the runtimes are significantly higher than those of the previous example for cases with equivalent numbers of source and target boxes.

Just as in the previous example, the smallest two problem sizes, $N = 16$ and $N = 32$, were observed to strong scale up to the limit of $p = N^3$ (respectively, 4096 and 32,768) processes. The most interesting problem size is $N = 64$, which corresponds to the largest problem which was able to fit in the memory of a single node. The strong scaling efficiency from one to 1024 nodes was observed to be 82.3% in this case, and, while this is significantly lower than that of the 2D hyperbolic Radon transform, it is arguably still impressive.

4. Conclusions and future work. A high-level parallelization of both general-purpose and analytical-interpolation based butterfly algorithms was presented, with the former resulting in a modeled runtime of

$$O\left(r^2 \frac{N^d}{p} \log N + \left(\beta r \frac{N^d}{p} + \alpha\right) \log p\right)$$

for d-dimensional problems with N^d source and target boxes, rank-r approximate interactions, and $p \leq N^d$ processes (with message latency, α, and inverse bandwidth, β). The key insight is that a careful manipulation of bitwise-partitions of the product space of the source and target domains can both keep the data (weight vectors) and the

computation (weight translations) evenly distributed, and teams of at most 2^d need interact via a reduce-scatter communication pattern at each of the $\log_2(N)$ stages of the algorithm. Algorithm 2.3 was then implemented in a black-box manner for kernels of the form $K(x,y) = \exp(i\Phi(x,y))$, and strong-scaling of 90.5% and 82.3% efficiency from one to 1024 nodes of Blue Gene/Q was observed for a 2D hyperbolic Radon transform and 3D generalized Radon analogue, respectively.

While, to the best of our knowledge, this is the first parallel implementation of the butterfly algorithm, there is still a significant amount of future work. The most straight-forward of which is to extend the current MPI implementation to exploit the N^d/p-fold trivial parallelism available for the local weight translations. This could potentially provide a significant performance improvement on "wide" architectures such as Blue Gene/Q. Though a significant amount of effort has already been devoted to improving the architectural efficiency, e.g., via batch evaluations of phase functions, further improvements are almost certainly waiting.

A more challenging direction involves the well-known fact that the butterfly algorithm can exploit sparse source and target domains [10, 34, 21], and so it would be worthwhile to extend our parallel algorithm into this regime. Lastly, the butterfly algorithm is closely related to the *directional Fast Multipole Method* [15], which makes use of low-rank interactions over spatial cones [22] which can be interpreted as satisfying the butterfly condition in angular coordinates. It would be interesting to investigate the degree to which our parallelization of the butterfly algorithm carries over to the directional FMM.

Availability. The distributed-memory implementation of the butterfly algorithm for kernels of the form $\exp(i\Phi(x,y))$, `DistButterfly`, is available at `http://github.com/poulson/dist-butterfly` under the GPLv3. All experiments in this manuscript made use of revision `f850f1691c`. Additionally, parallel implementations of interpolative decompositions are now available as part of Elemental [28], which is hosted at `http://code.google.com/p/elemental` under the New BSD License.

Acknowledgments. Jack Poulson would like to thank ALCF for providing access to Blue Gene/Q, and both Jeff Hammond and Hal Finkel for their generous help with both `C++11` support and performance issues on Blue Gene/Q. He would also like to thank Mark Tygert for detailed discussions on both the history and practical implementation of IDs and related factorizations, and Gregorio Quintana Ortí for discussions on high-performance RRQRs. This work was partially supported by NSF CAREER Grant No. 0846501 (L.Y.), DOE Grant No. DE-SC0009409 (L.Y.), and KAUST.

REFERENCES

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, *LogGP: Incorporating long messages into the LogP model for parallel computation*, J. Parallel and Dist. Comput., 44 (1997), no. 1, pp. 71–79.

[2] M. Barnett, S. Gupta, D. Payne, L. Shuler, and R. van de Geijn, *Interprocessor collective communication library (InterCom)*, Proc. Scalable High-Perf. Comput. Conf., 1994, pp. 357–364.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in numerical linear algebra*, Tech. Report, University of California at Berkeley, UCB/EECS-2011-15 (2011).

[4] G. Beylkin, *The inversion problem and applications of the generalized Radon transform*, Comm. Pure Appl. Math., 37 (1984), pp. 579–599.

[5] P. Businger and G. Golub, *Linear least squares solutions by Householder transformations*, Numer. Math., 7 (1965), pp. 269–276.
[6] E. Candès, L. Demanet, and L. Ying, *Fast computation of Fourier integral operators*, SIAM J. Sci. Comput., 29 (2007), no. 6, pp. 2464–2493.
[7] E. Candès, L. Demanet, and L. Ying, *A fast butterfly algorithm for the computation of Fourier integral operators*, SIAM J. Multiscale Modeling and Simulation, 7 (2009), no. 4, pp. 1727–1750.
[8] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, *Collective communication: theory, practice, and experience*, Concurrency and Comp.: Practice and Experience, 19 (2007), no. 13, pp. 1749–1783.
[9] S. Chandrasekaran and I. Ipsen, *On rank-revealing QR factorizations*, SIAM J. Matrix Anal. Appl., 15 (1994), no. 2, pp. 592–622.
[10] W. Chew and J. Song, *Fast Fourier transform of sparse spatial data to sparse Fourier data*, Antennas and Propagation Society International Symposium, 4 (2000), no. 4, pp. 2324–2327.
[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: towards a realistic model of parallel computation*, Proc. ACM SIGPLAN PPoPP, 1993, pp. 1–12.
[12] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.-K. Yeung, and R. Vuduc, *On the communication complexity of 3D FFTs and its implications for exascale*, Proc. NSF/TCPP Wkshp. Parallel and Dist. Comp. Edu., Shanghai, China 2012.
[13] L. Demanet, M. Ferrara, N. Maxwell, J. Poulson, and L. Ying, *A butterfly algorithm for synthetic aperture radar imaging*, SIAM J. Imaging Sciences, 5 (2012), no. 1, pp. 203–243.
[14] J. Demmel, L. Grigori, M. Gu, and H. Xiang, *Communication Avoiding Rank Revealing QR factorization with column pivoting*, Tech. Report, University of Tennessee, LAWN 276, 2013.
[15] B. Engquist and L. Ying, *Fast directional multilevel algorithms for oscillatory kernels*, SIAM J. Sci. Comput., 29 (2007), no. 4, pp. 1710–1737.
[16] I. Foster and P. Worley, *Parallel algorithms for the spectral transform method*, SIAM J. Sci. Comput., 18 (1997), no. 3, pp. 806–837.
[17] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comput. Phys., 73 (1987), pp. 325–348.
[18] M. Gu and S. Eisenstat, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM J. Sci. Comput., 17 (1996), pp. 848–869.
[19] R. Hockney, *The communication challenge for MPP: Intel Paragon and Meiko CS-2*, J. Parallel Comput., 20 (1994), no. 3, pp. 389–398.
[20] J. Hu, S. Fomel, L. Demanet, and L. Ying, *A fast butterfly algorithm for generalized Radon transforms*, Geophysics, To appear.
[21] S. Kunis and I. Melzer, *A stable and accurate butterfly sparse Fourier transform*, SIAM J. Numer. Anal., 50 (2012), pp. 1777–1800.
[22] P.-G. Martinsson and V. Rokhlin, *A fast direct solver for scattering problems involving elongated structures*, J. Comput. Phys., 221 (2007), no. 1, pp. 288–302.
[23] P.-G. Martinsson, V. Rokhlin, Y. Shkolnisky, and M. Tygert, *ID: A software package for low-rank approximation of matrices via interpolative decompositions, Version 0.2*, `http://cims.nyu.edu/~tygert/software.html`, 2008.
[24] E. Michielssen and A. Boag, *A multilevel matrix decomposition algorithm for analyzing scattering from large structures*, IEEE Trans. Antennas and Propagation, 44 (1996), no. 8, pp. 1086–1093.
[25] M. O'Neil, F. Woolfe, and V. Rokhlin, *An algorithm for the rapid evaluation of special function transforms*, Applied and Computational Harmonic Analysis, 28 (2010), no. 2, pp. 203–226.
[26] M. Pippig, *PFFT - An extension of FFTW to massively parallel architectures*, SIAM J. Sci. Comput., To appear (2013).
[27] M. Pippig and D. Potts, *Parallel three-dimensional nonequispaced Fast Fourier Transforms and their application to particle simulation*, Tech. Report, Chemnitz University of Technology, Preprint 8 (2012).
[28] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero, *Elemental: A new framework for distributed memory dense matrix computations*, ACM TOMS, 39 (2013), pp. 13:1–13:24.
[29] V. Rokhlin, *Rapid solution of integral equations of scattering theory in two dimensions*, J. Comput. Phys., 86 (1990), no. 2, pp. 414–439.
[30] R. Thakur, R. Rabenseifner, and W. Gropp, *Optimization of collective communication*

operations in MPICH, Int'l J. High Perf. Comput. Appl., 19 (2005), no. 1, pp. 49–66.

[31] M. Tygert, *Fast algorithms for spherical harmonic expansions, III*, J. Comput. Phys., 229 (2010), no. 18, pp. 6181–6192.

[32] F. van Zee, T. Smith, F. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. Meng Low, B. Marker, L. Killough, and R. van de Geijn, *Implementing Level-3 BLAS with BLIS: Early experience*, Tech. Report, University of Texas at Austin, TR-13-03 (2013).

[33] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, *A fast randomized algorithm for the approximation of matrices*, Appl. and Comp. Harmonic Anal., 25 (2008), no. 3, pp. 335–366.

[34] L. Ying, *Sparse Fourier transforms via butterfly algorithm*, SIAM J. Sci. Comput., 31 (2009), no. 3, pp. 1678–1694.