

# A Parallel Fast Algorithm for Applying Fourier Integral Operators

Jack Poulson<sup>1</sup>  
Laurent Demanet<sup>2</sup>   Nicholas Maxwell<sup>3</sup>   Lexing Ying<sup>4</sup>

<sup>1</sup>ICES, UT Austin

<sup>2</sup>Department of Mathematics, MIT

<sup>3</sup>Department of Mathematics, University of Houston

<sup>4</sup>Department of Mathematics, UT Austin

SIAM CSE, 2011 (draft)

# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - Results
  - ButterflyFIO

# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - Results
  - ButterflyFIO

# Fourier Integral Operators (FIOs)

The general form is

$$(Af)(x) = \int_{\mathbb{R}^d} a(x, k) e^{i\Phi(x, k)} \hat{f}(k) dk,$$

where  $\hat{f}$  is the Fourier transform of  $f$ .



# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Basic properties<sup>1</sup>:

- $a(x, k)$  is complex-valued. It is the *symbol* or *amplitude function*.
- $\Phi(x, k)$  is real-valued and is called the *phase function*.

---

<sup>1</sup>We lift many technicalities. Most notably, we do not require  $\Phi(x, k)$  to be homogeneous in  $k$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Basic properties<sup>1</sup>:

- $a(x, k)$  is complex-valued. It is the *symbol* or *amplitude function*.
- $\Phi(x, k)$  is real-valued and is called the *phase function*.

---

<sup>1</sup>We lift many technicalities. Most notably, we do not require  $\Phi(x, k)$  to be homogeneous in  $k$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Basic properties<sup>1</sup>:

- $a(x, k)$  is complex-valued. It is the *symbol* or *amplitude function*.
- $\Phi(x, k)$  is real-valued and is called the *phase function*.

---

<sup>1</sup>We lift many technicalities. Most notably, we do not require  $\Phi(x, k)$  to be homogeneous in  $k$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Basic properties<sup>1</sup>:

- $a(x, k)$  is complex-valued. It is the *symbol* or *amplitude function*.
- $\Phi(x, k)$  is real-valued and is called the *phase function*.

---

<sup>1</sup>We lift many technicalities. Most notably, we do not require  $\Phi(x, k)$  to be homogeneous in  $k$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Convenient definitions:

- When  $a(x, k) \equiv 1$ , an FIO reduces to an *Egorov operator*.
- Define  $A_R$  such that  $A = A_R \circ \mathcal{F}$ , where  $\mathcal{F}$  is the Fourier transform. (We also have  $A^* = \mathcal{F}^* \circ A_R^* = \mathcal{F}^{-1} \circ A_R^*$ .)
- We will simply refer to  $A_R$  as a *reduced FIO* (RFIO) since it will be our main focus.

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Convenient definitions:

- When  $a(x, k) \equiv 1$ , an FIO reduces to an *Egorov operator*.
- Define  $A_R$  such that  $A = A_R \circ \mathcal{F}$ , where  $\mathcal{F}$  is the Fourier transform. (We also have  $A^* = \mathcal{F}^* \circ A_R^* = \mathcal{F}^{-1} \circ A_R^*$ .)
- We will simply refer to  $A_R$  as a *reduced FIO* (RFIO) since it will be our main focus.

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Convenient definitions:

- When  $a(x, k) \equiv 1$ , an FIO reduces to an *Egorov operator*.
- Define  $A_R$  such that  $A = A_R \circ \mathcal{F}$ , where  $\mathcal{F}$  is the Fourier transform. (We also have  $A^* = \mathcal{F}^* \circ A_R^* = \mathcal{F}^{-1} \circ A_R^*$ .)
- We will simply refer to  $A_R$  as a *reduced FIO* (RFIO) since it will be our main focus.



# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

Convenient definitions:

- When  $a(x, k) \equiv 1$ , an FIO reduces to an *Egorov operator*.
- Define  $A_R$  such that  $A = A_R \circ \mathcal{F}$ , where  $\mathcal{F}$  is the Fourier transform. (We also have  $A^* = \mathcal{F}^* \circ A_R^* = \mathcal{F}^{-1} \circ A_R^*$ .)
- We will simply refer to  $A_R$  as a *reduced FIO* (RFIO) since it will be our main focus.

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

The discrete RFIO is then simply

$$(A_R g)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} g(k_j).$$

Notice that the continuous Fourier transform is an RFIO where  $a(x, k) \equiv (2\pi)^{-d/2}$  and  $\Phi(x, k) = -2\pi x \cdot k$ .

# Fourier Integral Operators (FIOs)

Our discrete form is

$$(Af)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} \hat{f}(k_j),$$

where  $\hat{f}$  is the Fourier transform of  $f$ .

The discrete RFIO is then simply

$$(A_R g)(x_i) = \sum_{k_j \in \mathbb{R}^d} a(x_i, k_j) e^{i\Phi(x_i, k_j)} g(k_j).$$

Notice that the continuous Fourier transform is an RFIO where  $a(x, k) \equiv (2\pi)^{-d/2}$  and  $\Phi(x, k) = -2\pi x \cdot k$ .

# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - Results
  - ButterflyFIO

## $\epsilon$ -separation Rank

Suppose  $a : \Omega_x \times \Omega_k \rightarrow \mathbb{C}$  and  $\Phi : \Omega_x \times \Omega_k \rightarrow \mathbb{R}$  are smooth. Then, for a given  $\epsilon > 0$ , there exists  $d_\epsilon > 0$  and  $r_\epsilon \in \mathbb{N}$  such that, for any  $X \subset \Omega_x$  and  $K \subset \Omega_k$  satisfying  $w(X)w(K) \leq d_\epsilon$ ,

$$\left| a(x, k) e^{i\Phi(x, k)} - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \beta_t^{XK}(k) \right| \leq \epsilon, \quad \forall x \in X, \forall k \in K.$$

Summation over each  $g(k)$  yields

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon,$$

which can be rewritten as

$$(A_R g_K)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X, \forall k \in K.$$

# $\epsilon$ -separation Rank

Suppose  $a : \Omega_x \times \Omega_k \rightarrow \mathbb{C}$  and  $\Phi : \Omega_x \times \Omega_k \rightarrow \mathbb{R}$  are smooth. Then, for a given  $\epsilon > 0$ , there exists  $d_\epsilon > 0$  and  $r_\epsilon \in \mathbb{N}$  such that, for any  $X \subset \Omega_x$  and  $K \subset \Omega_k$  satisfying  $w(X)w(K) \leq d_\epsilon$ ,

$$\left| a(x, k) e^{i\Phi(x, k)} - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \beta_t^{XK}(k) \right| \leq \epsilon, \quad \forall x \in X, \forall k \in K.$$

Summation over each  $g(k)$  yields

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon,$$

which can be rewritten as

$$(A_R g_K)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X, \forall k \in K.$$

# $\epsilon$ -separation Rank

Suppose  $a : \Omega_x \times \Omega_k \rightarrow \mathbb{C}$  and  $\Phi : \Omega_x \times \Omega_k \rightarrow \mathbb{R}$  are smooth. Then, for a given  $\epsilon > 0$ , there exists  $d_\epsilon > 0$  and  $r_\epsilon \in \mathbb{N}$  such that, for any  $X \subset \Omega_x$  and  $K \subset \Omega_k$  satisfying  $w(X)w(K) \leq d_\epsilon$ ,

$$\left| a(x, k) e^{i\Phi(x, k)} - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \beta_t^{XK}(k) \right| \leq \epsilon, \quad \forall x \in X, \forall k \in K.$$

Summation over each  $g(k)$  yields

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon,$$

which can be rewritten as

$$(A_R g_K)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X, \forall k \in K.$$

# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_X$  and  $K = \Omega_K$  are both allowed, just not at once.
- If we could cover  $\Omega_X$  with many low-rank approx's using  $K = \Omega_K$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_K$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_X$ .
- If  $X = \Omega_X$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_X, K)$  interactions into many  $(X, \Omega_K)$  interactions...



# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_x$  and  $K = \Omega_k$  are both allowed, just not at once.
- If we could cover  $\Omega_x$  with many low-rank approx's using  $K = \Omega_k$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_k$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_x$ .
- If  $X = \Omega_x$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_x, K)$  interactions into many  $(X, \Omega_k)$  interactions...

# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_X$  and  $K = \Omega_K$  are both allowed, just not at once.
- If we could cover  $\Omega_X$  with many low-rank approx's using  $K = \Omega_K$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_K$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_X$ .
- If  $X = \Omega_X$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_X, K)$  interactions into many  $(X, \Omega_K)$  interactions...

# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_X$  and  $K = \Omega_K$  are both allowed, just not at once.
- If we could cover  $\Omega_X$  with many low-rank approx's using  $K = \Omega_K$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_K$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_X$ .
- If  $X = \Omega_X$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_X, K)$  interactions into many  $(X, \Omega_K)$  interactions...

# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_X$  and  $K = \Omega_K$  are both allowed, just not at once.
- If we could cover  $\Omega_X$  with many low-rank approx's using  $K = \Omega_K$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_K$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_X$ .
- If  $X = \Omega_X$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_X, K)$  interactions into many  $(X, \Omega_K)$  interactions...

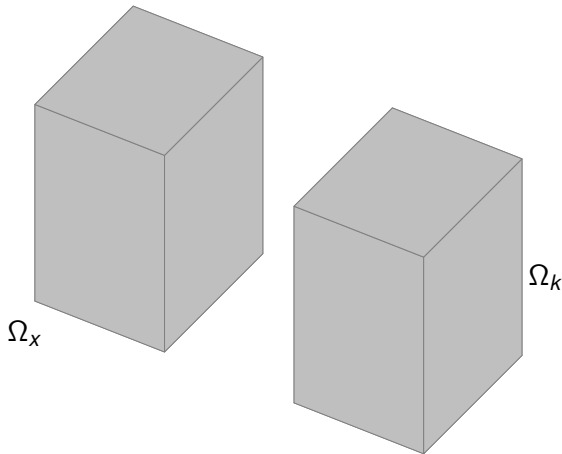
# Using the $\epsilon$ -separation Rank

$$\left| (A_R g_K)(x) - \sum_{t=0}^{r_\epsilon} \alpha_t^{XK}(x) \left( \sum_{k \in K} \beta_t^{XK}(k) g(k) \right) \right| \leq \left( \sum_{k \in K} |g(k)| \right) \epsilon, \quad \forall x \in X, \forall k \in K.$$

- $w(X)w(K) \leq d_\epsilon$  only constrains product of diameters of  $X$  and  $K$ .
- $X = \Omega_X$  and  $K = \Omega_K$  are both allowed, just not at once.
- If we could cover  $\Omega_X$  with many low-rank approx's using  $K = \Omega_K$ , we could still cheaply evaluate  $(A_R g)(x)$ .
- But if  $K = \Omega_K$ , forming  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is expensive and must be done for each  $X \subsetneq \Omega_X$ .
- If  $X = \Omega_X$ , so that  $K$  is small, forming each  $\sum_{k \in K} \beta_t^{XK}(k) g(k)$  is cheap!
- Then transform many  $(\Omega_X, K)$  interactions into many  $(X, \Omega_K)$  interactions...

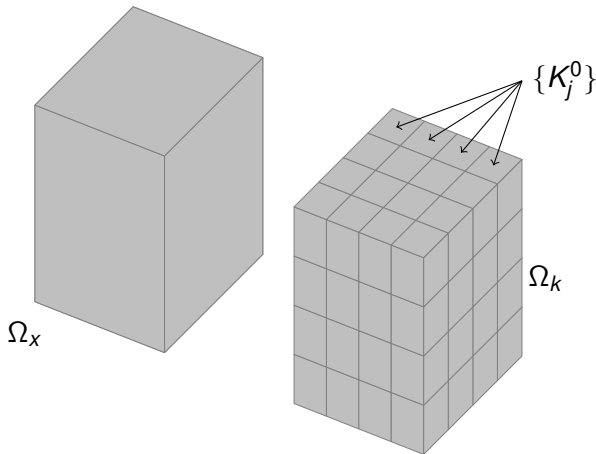
## Butterfly Algorithm Example: Level 0

Choose  $N = 2^m$ ,  $m \in \mathbb{N}$ , such that  $w(\Omega_x)w(\Omega_k)/N \leq d_\epsilon$ .



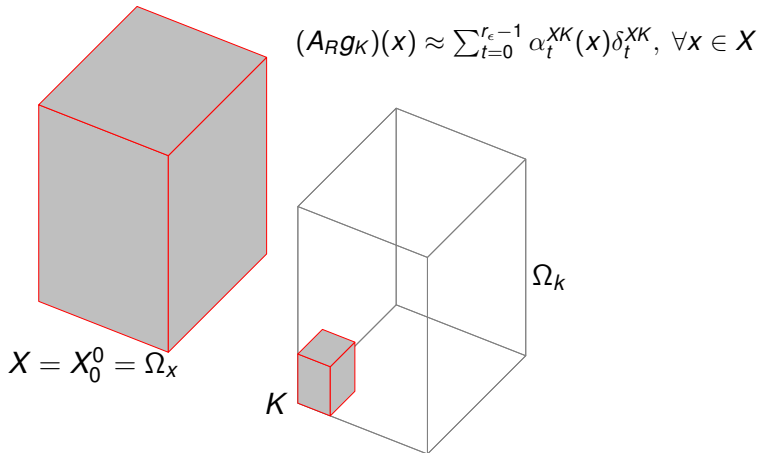
# Butterfly Algorithm Example: Level 0

Partition  $\Omega_k$  into  $N$  pieces in each dimension (i.e.,  $N = 4$ ),  $\{K_j^0\}_{j=0}^{N^d-1}$ .



# Butterfly Algorithm Example: Level 0

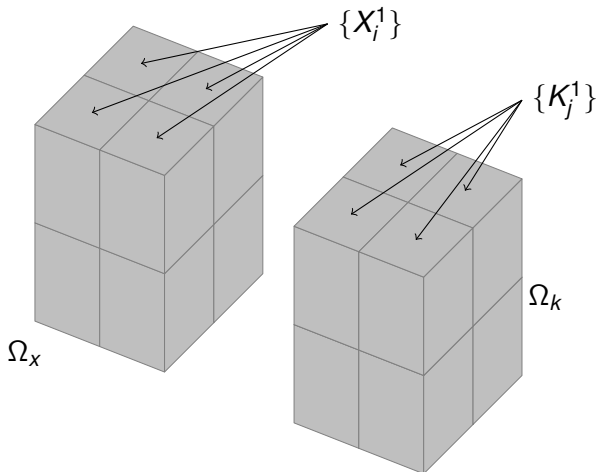
For each  $K = K_j^0$ , initialize  $\{\delta_t^{XK}\}$  from sources in  $K$ .





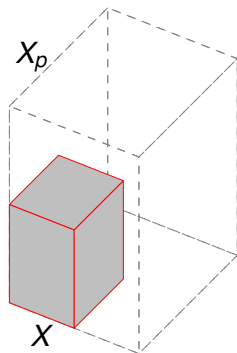
# Butterfly Algorithm Example: Level 1

Refine  $\Omega_x$  and coarsen  $\Omega_k$  by factor of 2 to form  $\{X_i^1\}$  and  $\{K_j^1\}$ .

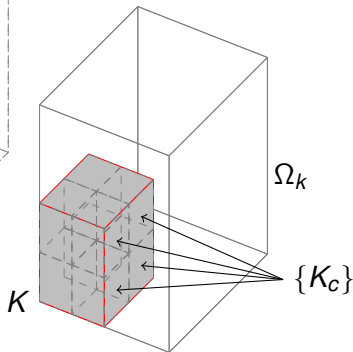


# Butterfly Algorithm Example: Level 1

For each  $X = X_i^1$  and  $K = K_j^1$ , form  $\{\delta_t^{XK}\}$  by combining the weights for the interaction of  $K$ 's  $2^d$  children with  $X$ 's parent.

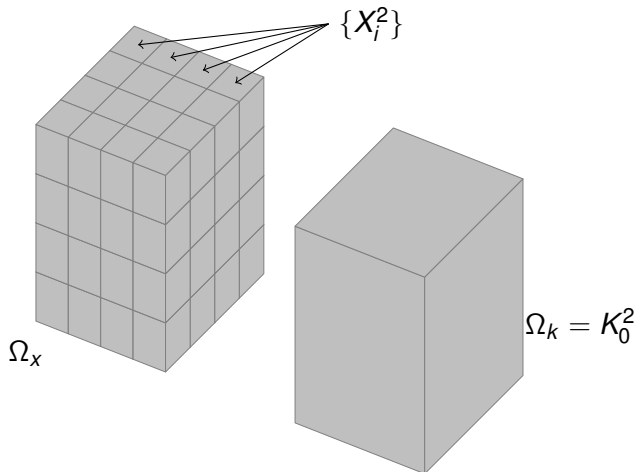


$$(A_{RGK})(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X$$



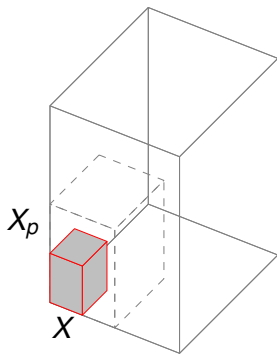
## Butterfly Algorithm Example: Level 2

Refine  $\Omega_x$  and coarsen  $\Omega_k$  by factor of 2 to form  $\{X_i^2\}$  and  $\{K_j^2\}$ .

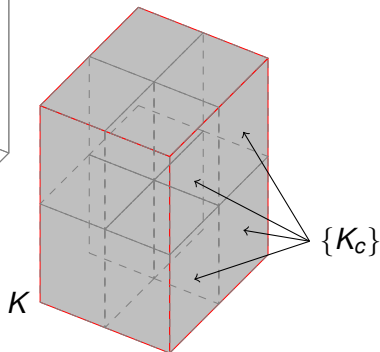


## Butterfly Algorithm Example: Level 2

For each  $X = X_j^2$ , form  $\{\delta_t^{XK}\}$  by combining the weights for the interaction of  $K$ 's  $2^d$  children with  $X$ 's parent.



$$(A_{RGK})(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X$$



# Butterfly Algorithm Example: Summary

- After completion,  $K = \Omega_k$ , so  $g_K = g$  and we have

$$(A_R g)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X.$$

- In general, there are  $\log_2(N)$  levels.
- Since  $\{X_i^{\log_2(N)}\}$  covers  $\Omega_x$ , one can evaluate  $(A_R g)(x)$  for any  $x \in \Omega_x$  with  $\mathcal{O}(1)$  work.
- Each level involves  $N^d$  interactions requiring  $\mathcal{O}(1)$  work each.
- Complexity is  $\mathcal{O}(N^d \log_2(N))$ , vs.  $\mathcal{O}(N^{2d})$  naïve complexity.

## Butterfly Algorithm Example: Summary

- After completion,  $K = \Omega_k$ , so  $g_K = g$  and we have

$$(A_R g)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X.$$

- In general, there are  $\log_2(N)$  levels.
- Since  $\{X_i^{\log_2(N)}\}$  covers  $\Omega_x$ , one can evaluate  $(A_R g)(x)$  for any  $x \in \Omega_x$  with  $\mathcal{O}(1)$  work.
- Each level involves  $N^d$  interactions requiring  $\mathcal{O}(1)$  work each.
- Complexity is  $\mathcal{O}(N^d \log_2(N))$ , vs.  $\mathcal{O}(N^{2d})$  naïve complexity.

## Butterfly Algorithm Example: Summary

- After completion,  $K = \Omega_k$ , so  $g_K = g$  and we have

$$(A_R g)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X.$$

- In general, there are  $\log_2(N)$  levels.
- Since  $\{X_i^{\log_2(N)}\}$  covers  $\Omega_x$ , one can evaluate  $(A_R g)(x)$  for any  $x \in \Omega_x$  with  $\mathcal{O}(1)$  work.
- Each level involves  $N^d$  interactions requiring  $\mathcal{O}(1)$  work each.
- Complexity is  $\mathcal{O}(N^d \log_2(N))$ , vs.  $\mathcal{O}(N^{2d})$  naïve complexity.

## Butterfly Algorithm Example: Summary

- After completion,  $K = \Omega_k$ , so  $g_K = g$  and we have

$$(A_R g)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X.$$

- In general, there are  $\log_2(N)$  levels.
- Since  $\{X_i^{\log_2(N)}\}$  covers  $\Omega_x$ , one can evaluate  $(A_R g)(x)$  for any  $x \in \Omega_x$  with  $\mathcal{O}(1)$  work.
- Each level involves  $N^d$  interactions requiring  $\mathcal{O}(1)$  work each.
- Complexity is  $\mathcal{O}(N^d \log_2(N))$ , vs.  $\mathcal{O}(N^{2d})$  naïve complexity.



## Butterfly Algorithm Example: Summary

- After completion,  $K = \Omega_k$ , so  $g_K = g$  and we have

$$(A_R g)(x) \approx \sum_{t=0}^{r_\epsilon-1} \alpha_t^{XK}(x) \delta_t^{XK}, \quad \forall x \in X.$$

- In general, there are  $\log_2(N)$  levels.
- Since  $\{X_i^{\log_2(N)}\}$  covers  $\Omega_x$ , one can evaluate  $(A_R g)(x)$  for any  $x \in \Omega_x$  with  $\mathcal{O}(1)$  work.
- Each level involves  $N^d$  interactions requiring  $\mathcal{O}(1)$  work each.
- Complexity is  $\mathcal{O}(N^d \log_2(N))$ , vs.  $\mathcal{O}(N^{2d})$  naïve complexity.

# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - Results
  - ButterflyFIO

# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).

# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).

# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).

# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).

# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).

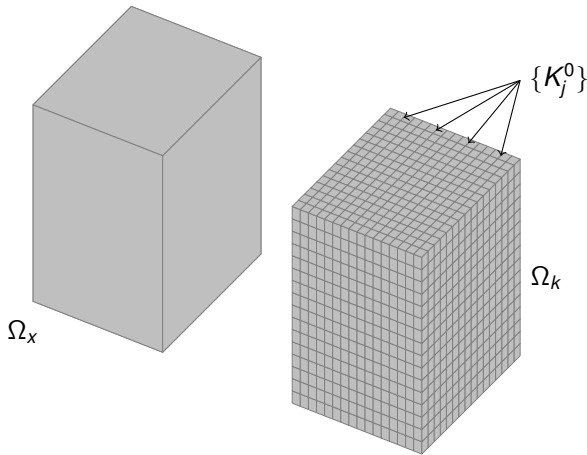
# Parallelization Strategy

- Grids change at every level, but the number of interactions is constant ( $N^d$ ).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with  $\Omega_k$  fully distributed and end with  $\Omega_x$  fully distributed.
- Use power of two number of processes so that repeated bisections can be used to distribute the domains.
- Gradually migrate the bisections from  $\Omega_k$  to  $\Omega_x$  during the coarsening of  $\Omega_k$ .
- Limit to at most  $N^d$  processes (one interaction per process).



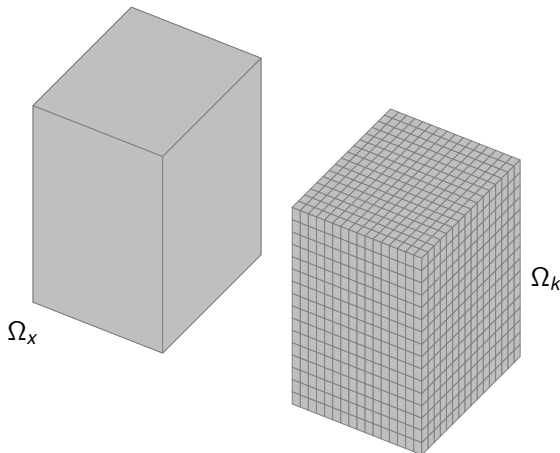
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We begin with  $\Omega_k$  partitioned into  $N^d$  pieces,  $\{K_j^0\}$ .



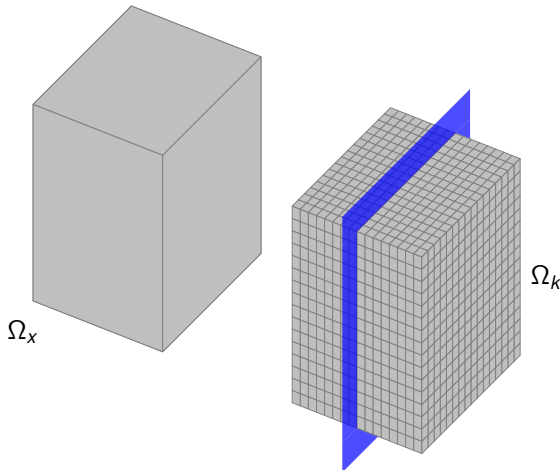
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We cyclically bisect  $\Omega_k \log_2(p)$  times over the  $d$  dimensions to distribute  $\Omega_k$ .



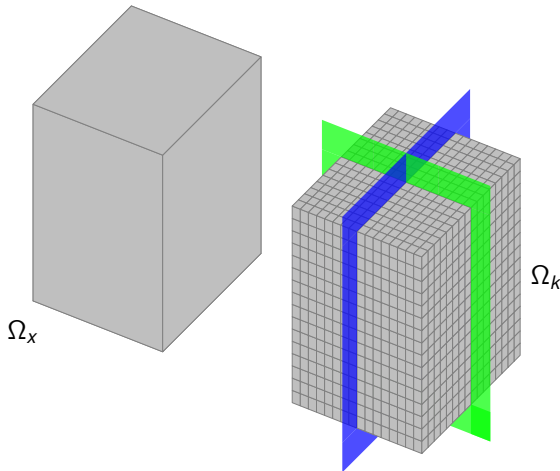
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We cyclically bisect  $\Omega_k \log_2(p)$  times over the  $d$  dimensions to distribute  $\Omega_k$ .



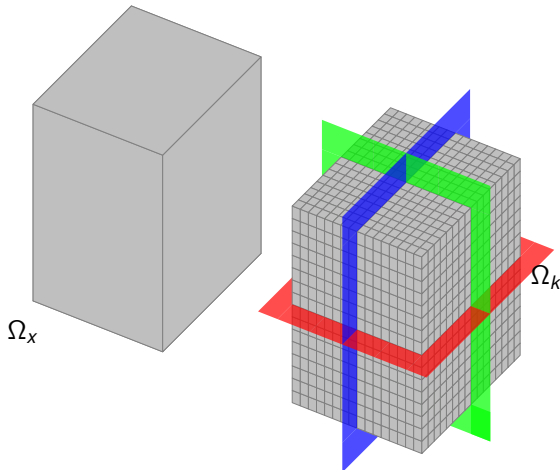
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We cyclically bisect  $\Omega_k \log_2(p)$  times over the  $d$  dimensions to distribute  $\Omega_k$ .



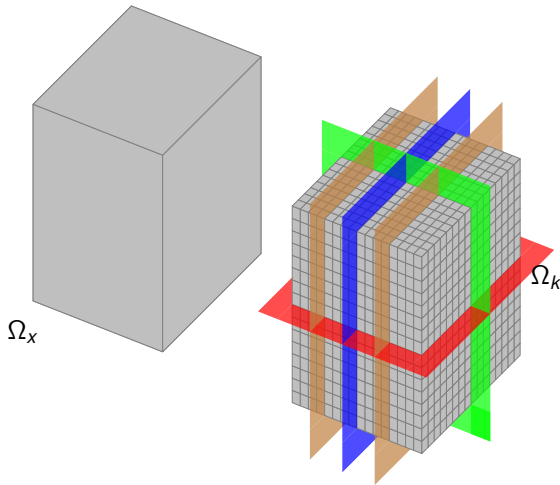
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We cyclically bisect  $\Omega_k \log_2(p)$  times over the  $d$  dimensions to distribute  $\Omega_k$ .



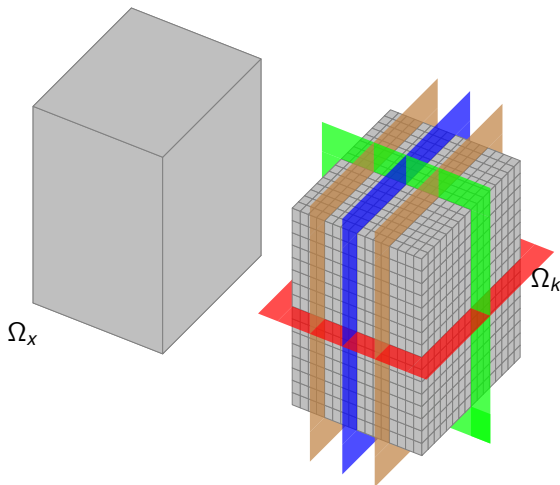
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

We cyclically bisect  $\Omega_k \log_2(p)$  times over the  $d$  dimensions to distribute  $\Omega_k$ .



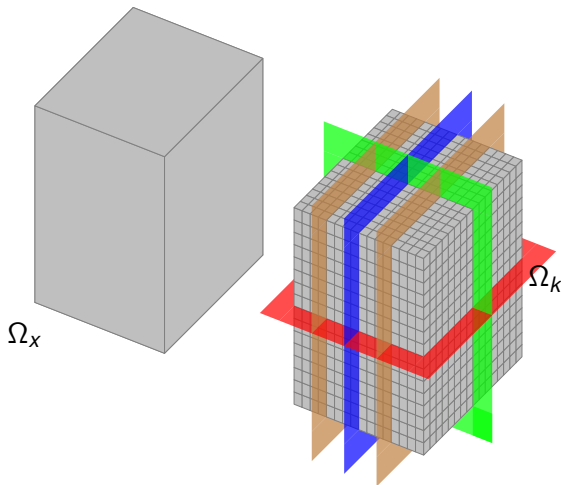
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 0

Each process then initializes  $N^d/p$  interactions between  $\Omega_x$  and each  $K \in \{K_j^0\}$  within its chunk of  $\Omega_k$ .



## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 1

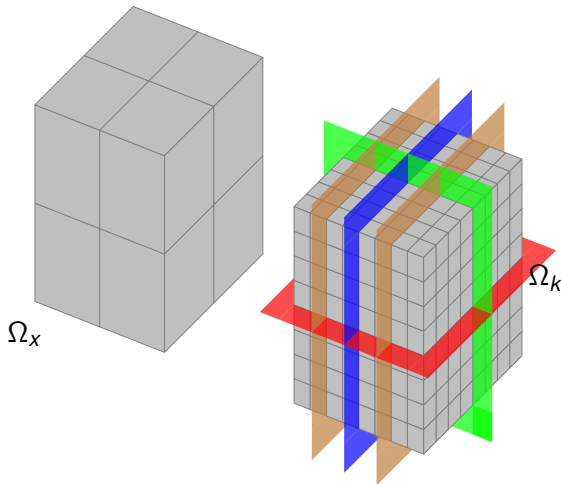
Refinement of  $\Omega_x$  and coarsening of  $\Omega_k$  is done locally on each process since no process boundaries are crossed.





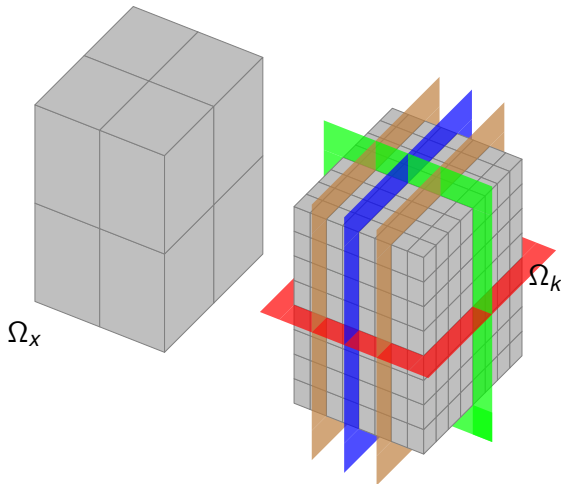
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 1

Refinement of  $\Omega_x$  and coarsening of  $\Omega_k$  is done locally on each process since no process boundaries are crossed.



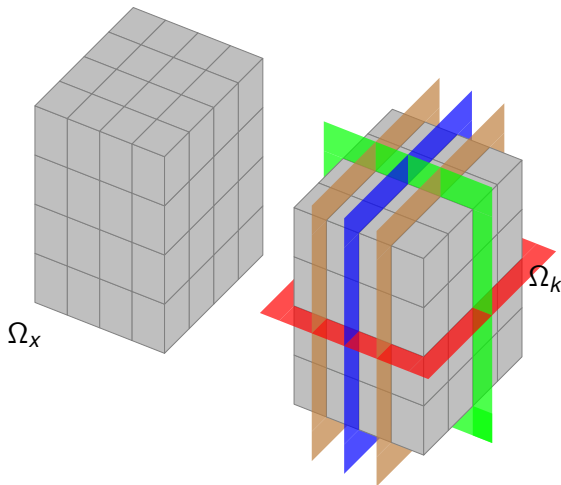
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 2

Refinement of  $\Omega_x$  and coarsening of  $\Omega_k$  is again local because no process boundaries are crossed.



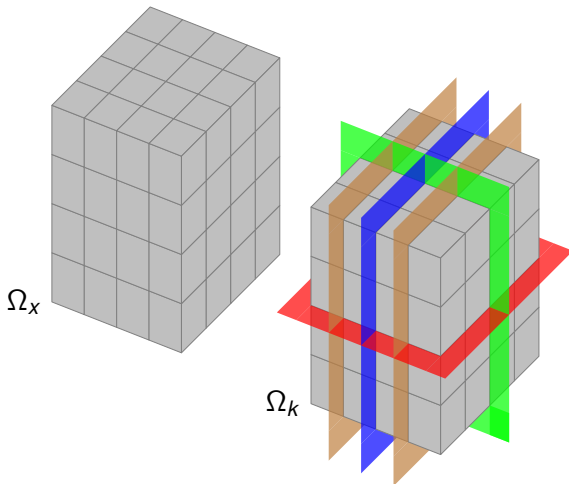
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 2

Refinement of  $\Omega_x$  and coarsening of  $\Omega_k$  is again local because no process boundaries are crossed.



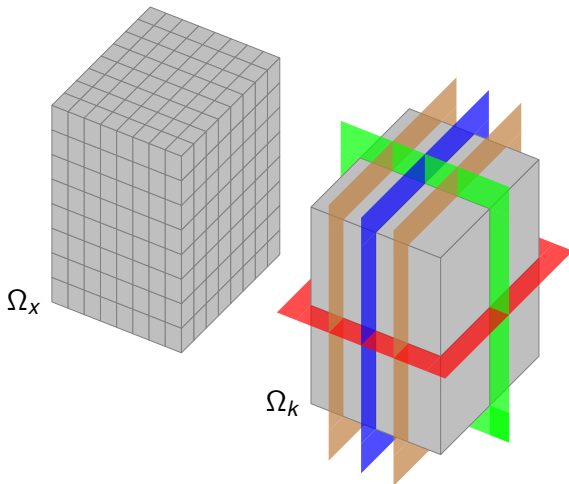
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 3

Coarsening  $\Omega_k$  crosses the fourth set of bisections (in brown).



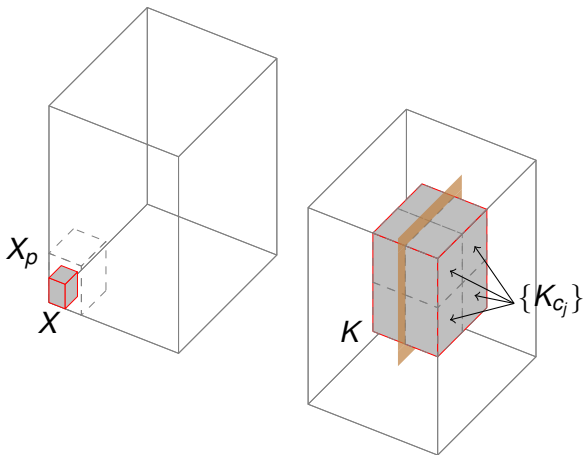
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 3

Coarsening  $\Omega_k$  crosses the fourth set of bisections (in brown).



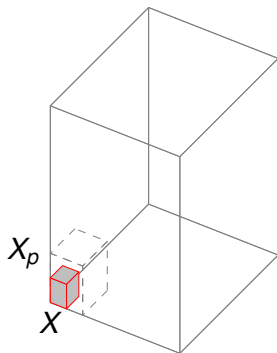
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 3

The  $\{\delta_t^{XK}\}$  weights are a sum of the contributions from each  $(X_p, K_{c_j})$ . **MPI\_Reduce** sums in teams of  $2^m$ ,  $m$  crossed parts.?

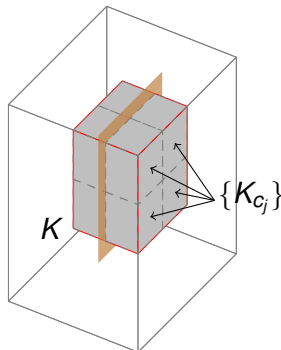


## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 3

The  $\{\delta_t^{XK}\}$  weights are a sum of the contributions from each  $(X_p, K_{c_j})$ . **MPI\_Reduce** sums in teams of  $2^m$ ,  $m$  crossed parts.?

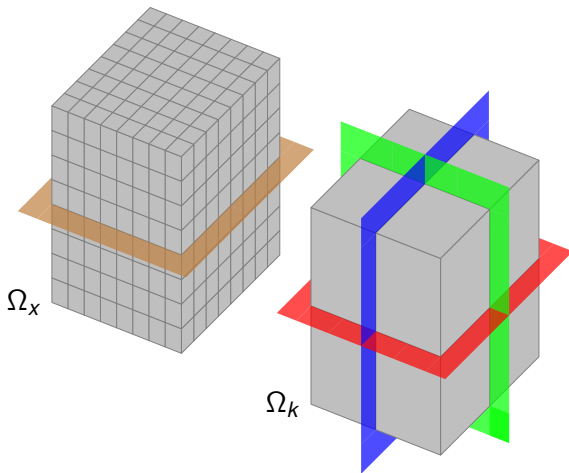


No. **MPI\_Reduce\_scatter** sum in teams of  $2^m$ .



## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 3

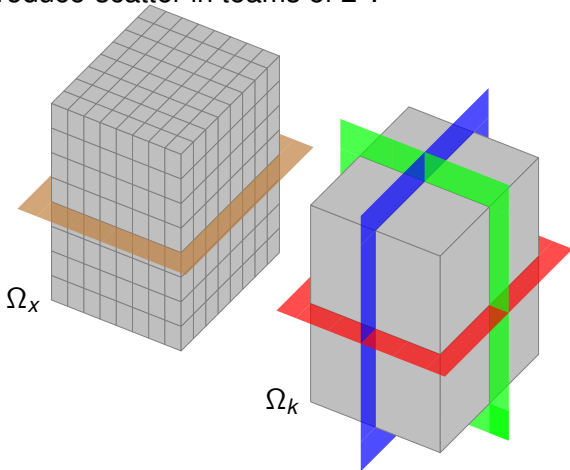
The **MPI\_Reduce\_scatter** is performed so that it effectively moves the crossed  $\Omega_k$  partitions onto  $\Omega_x$ .





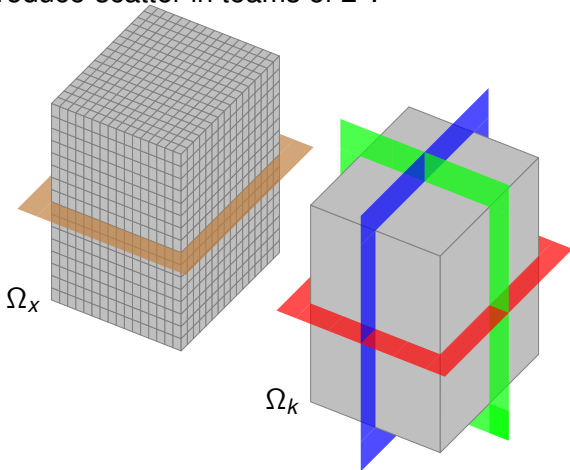
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 4

The last coarsening crosses 3 partition levels, so processes will reduce-scatter in teams of  $2^3$ .



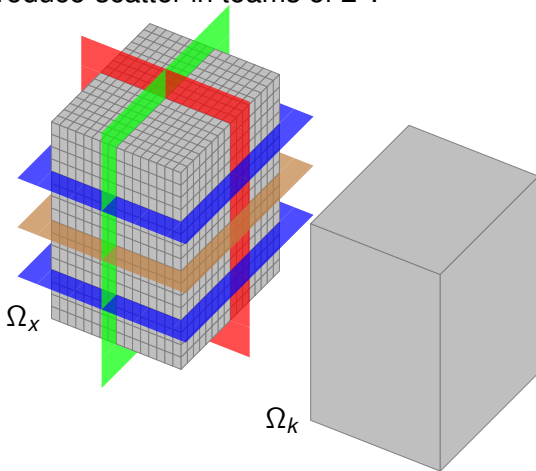
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 4

The last coarsening crosses 3 partition levels, so processes will reduce-scatter in teams of  $2^3$ .



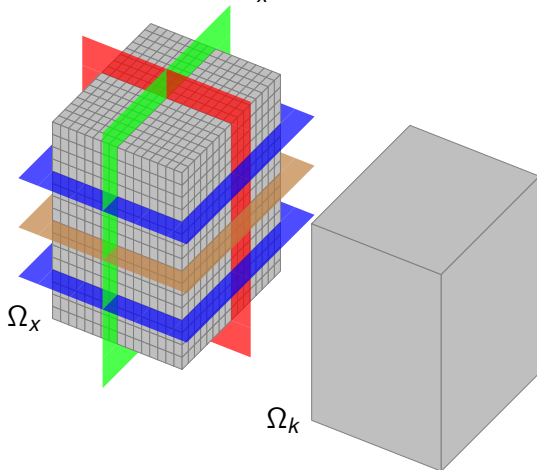
## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 4

The last coarsening crosses 3 partition levels, so processes will reduce-scatter in teams of  $2^3$ .



## Example for $d = 3$ , $N = 16$ , and $p = 16$ : Level 4

Done! Each process can now locally evaluate  $(A_R g)(x)$  for all  $x$  within its chunk of  $\Omega_x$ .



# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - **Results**
  - ButterflyFIO

# Blue Gene/P



# Blue Gene/P

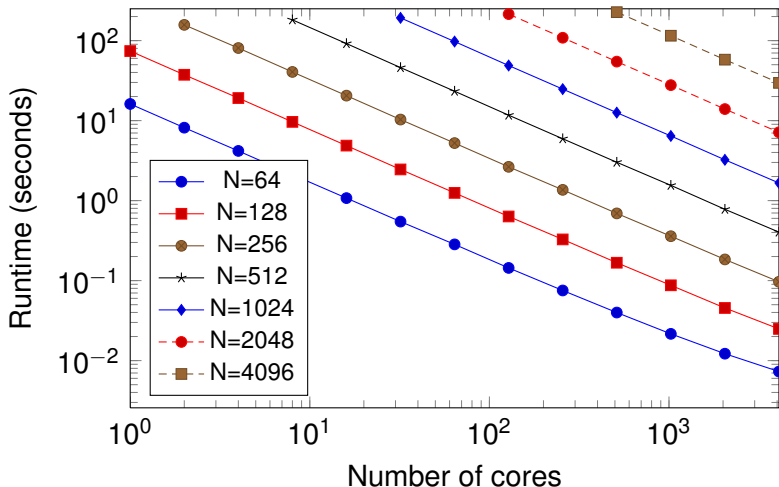


# Blue Gene/P

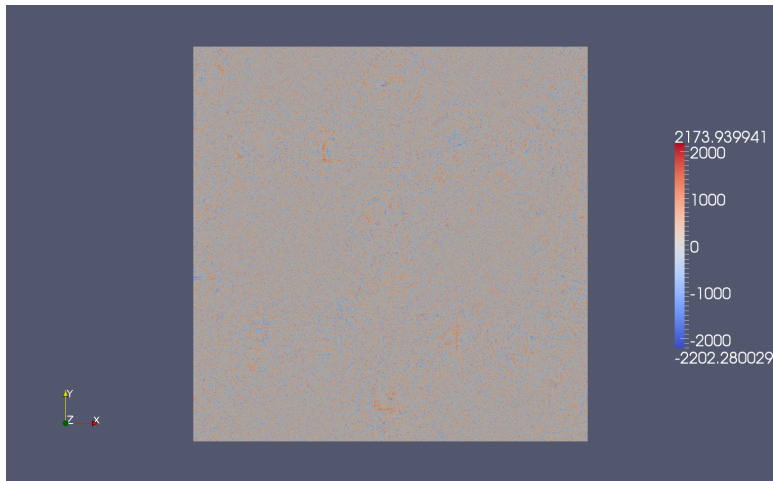




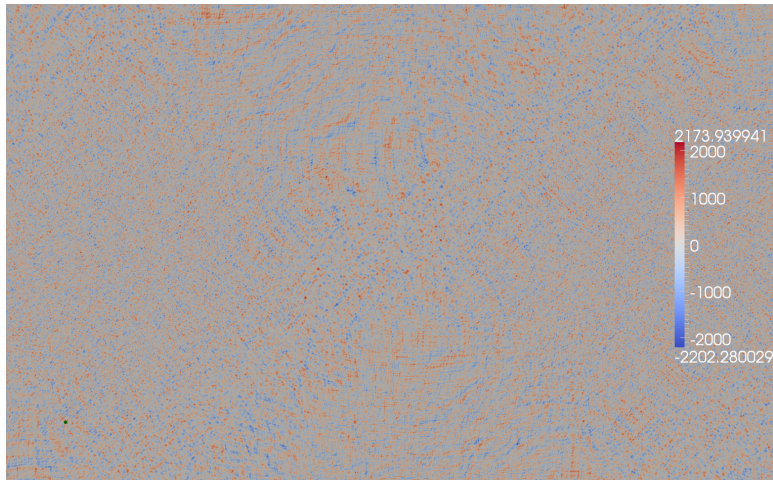
# Strong Scaling of Non-uniform 2d Generalized Radon Analogue (w/o Fourier Transform)



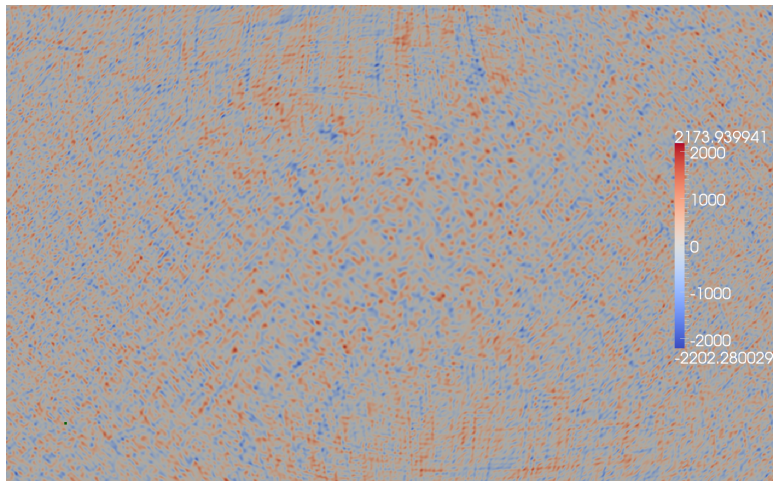
# Results from Generalized Radon Analogue with $N = 1024$



# Results from Generalized Radon Analogue with $N = 1024$



# Results from Generalized Radon Analogue with $N = 1024$



# Outline

- 1 Background
  - FIOs and RFIOs
  - The Butterfly Algorithm for Applying RFIOs
- 2 Our Contribution
  - Parallelizing the Butterfly Algorithm
  - Results
  - **ButterflyFIO**

# ButterflyFIO

- Available at <http://code.google.com/p/butterflyfio>
- It is licensed under GPLv3.
- Non-uniform RFIOs and Fourier transforms already supported.
- Templated, dimension-independent, header-only C++.
- Amplitude and phase functions can simply be passed in.
- Its dependencies are MPI, BLAS, and LAPACK.
- CMake build system.
- Can easily dump potential fields into parallel VTK files.
- Contributions are welcome!

# Summary

- **New parallel method** for the butterfly algorithm that easily strong scales to tens of thousands of cores.
- First massively parallel fast FIO implementation.
- First massively parallel non-uniform fast Fourier transform implementation?
- Source code available at <http://code.google.com/p/butterflyfio>