A Parallel Fast Algorithm for Applying Fourier Integral Operators

Jack Poulson[1]
Laurent Demanet[2] Nicholas Maxwell[3] Lexing Ying[4]

[1]ICES, UT Austin

[2]Department of Mathematics, MIT

[3]Department of Mathematics, University of Houston

[4]Department of Mathematics, UT Austin

TCCS, 2011

Outline

Outline

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[1]:

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

[1] 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[1]:

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

[1] 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[1]:

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

[1] 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[1]:

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

[1] 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

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

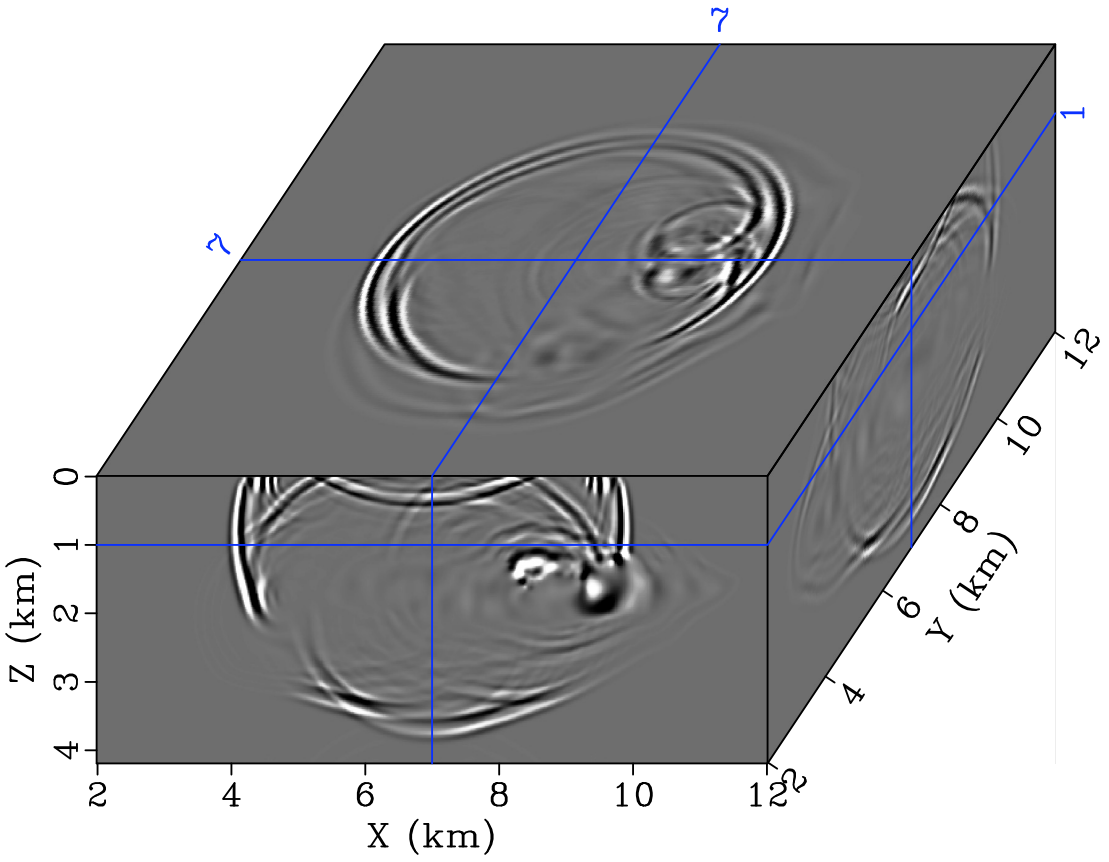
Notice that the continuous Fourier transform is an RFIO where $a(x,k) \equiv (2\pi)^{-d/2}$ and $\Phi(x,k) = -2\pi x \cdot k$.

Outline

Motivation

Wave extrapolation through successive Egorov operators:

$$P(x, t + \Delta t) \approx \int e^{i(x \cdot k + V(x,k)|k|\Delta t)} \hat{P}(k, t) dk$$



Outline

ϵ-separation Rank

If $a : \Omega_x \times \Omega_k \to \mathbb{C}$ and $\Phi : \Omega_x \times \Omega_k \to \mathbb{R}$ are sufficiently 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

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

ϵ-separation Rank

If $a : \Omega_x \times \Omega_k \to \mathbb{C}$ and $\Phi : \Omega_x \times \Omega_k \to \mathbb{R}$ are sufficiently 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

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

ϵ-separation Rank

If $a : \Omega_x \times \Omega_k \to \mathbb{C}$ and $\Phi : \Omega_x \times \Omega_k \to \mathbb{R}$ are sufficiently 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 ϵ-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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

Using the ϵ-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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

Using the ϵ-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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

Using the ϵ-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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

Using the ϵ-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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

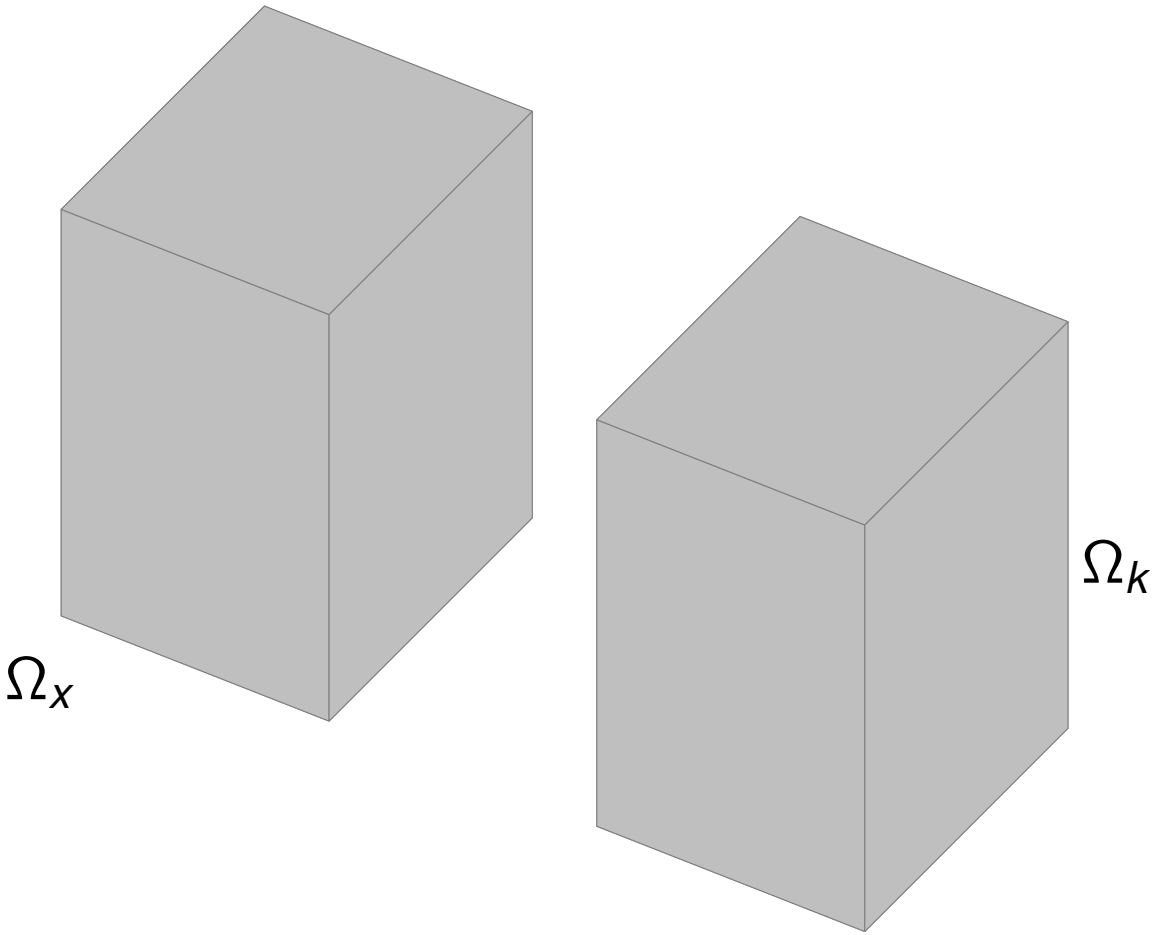
Using the ϵ-separation Rank

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

- $w(X)w(K) \leq d_\epsilon$ 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 Ω_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 (Ω_x, K) interactions into many (X, Ω_k) interactions...

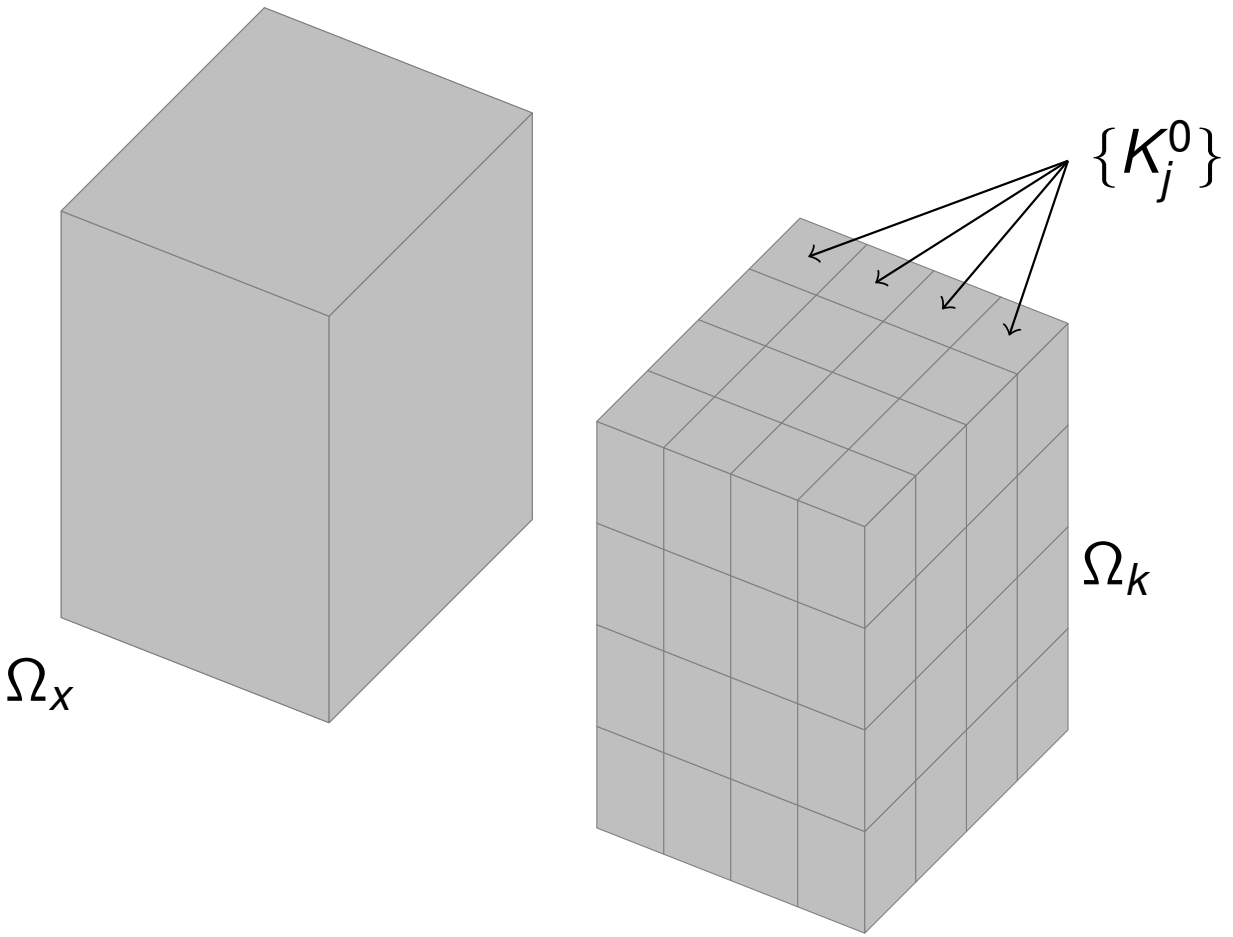
Butterfly Algorithm Example: Level 0

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



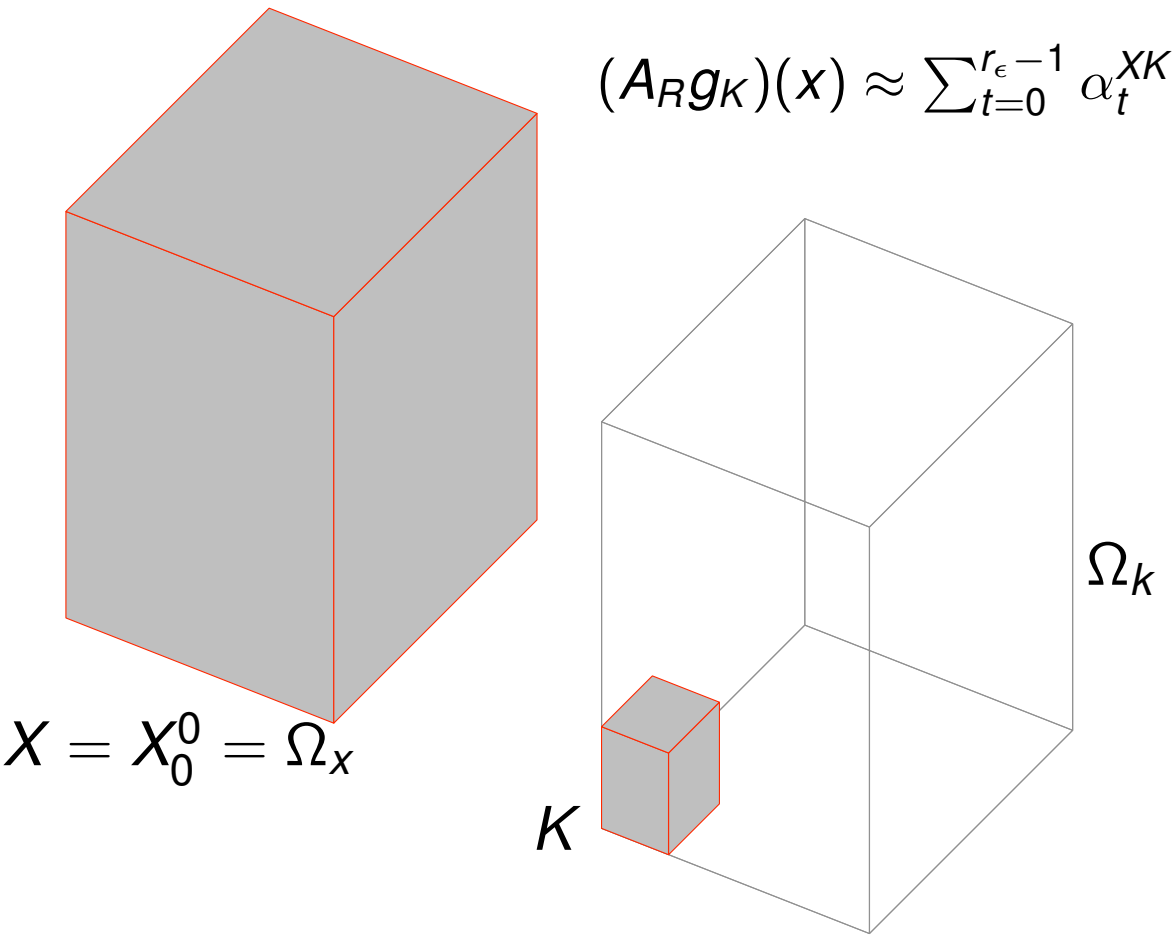
Butterfly Algorithm Example: Level 0

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



Butterfly Algorithm Example: Level 0

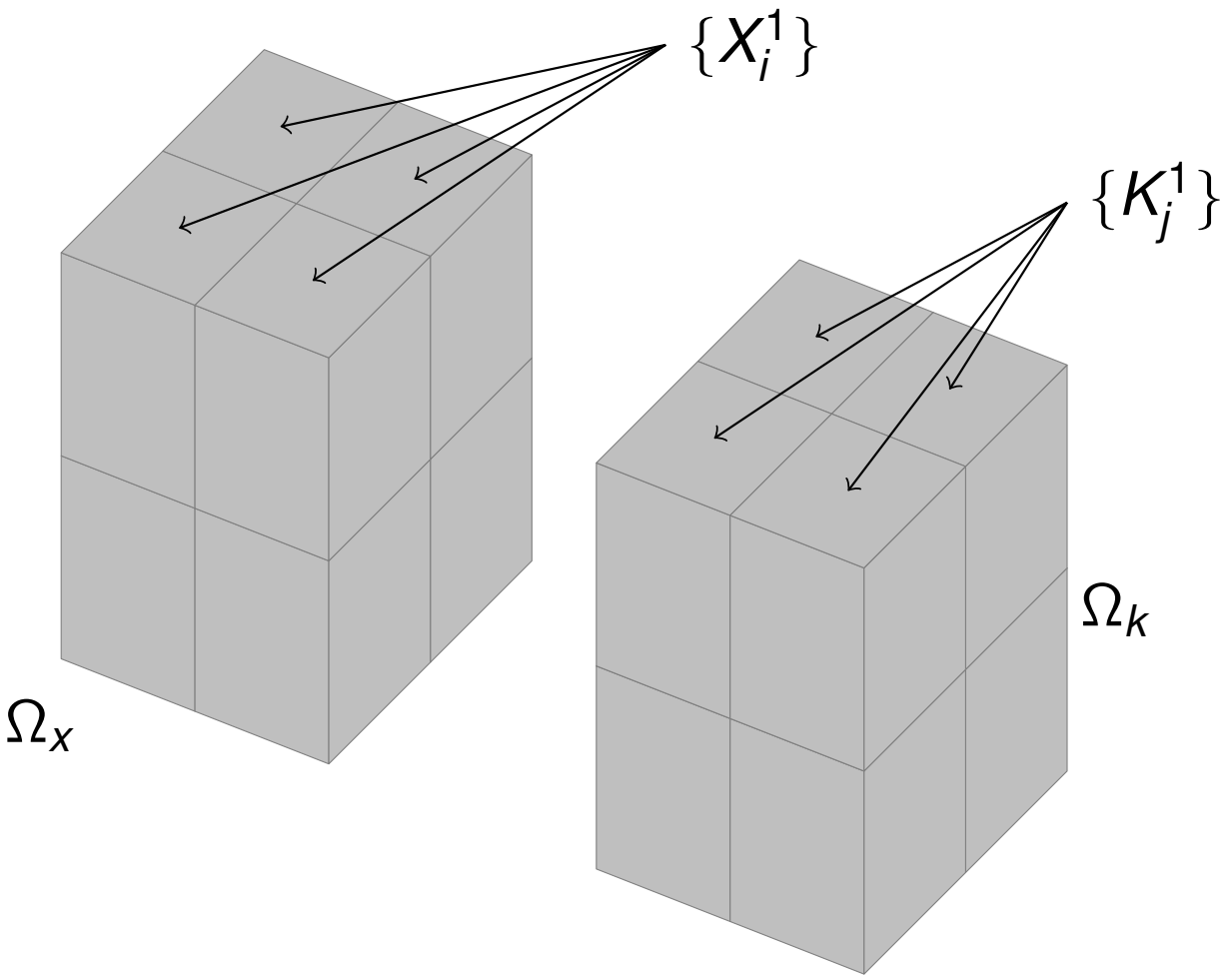
For each $K = K_j^0$, initialize $\{\delta_t^{XK}\}$ from sources in K.



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

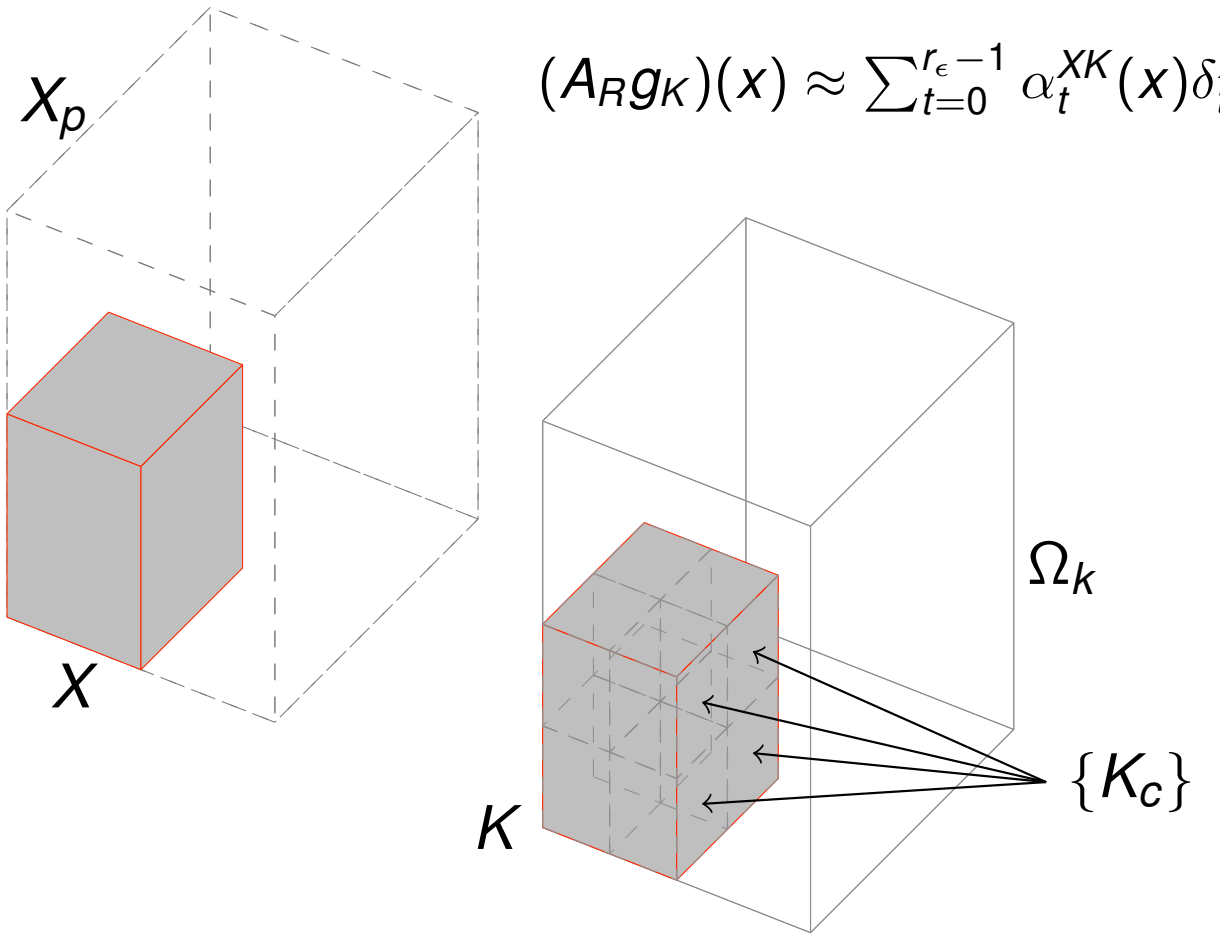
Butterfly Algorithm Example: Level 1

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



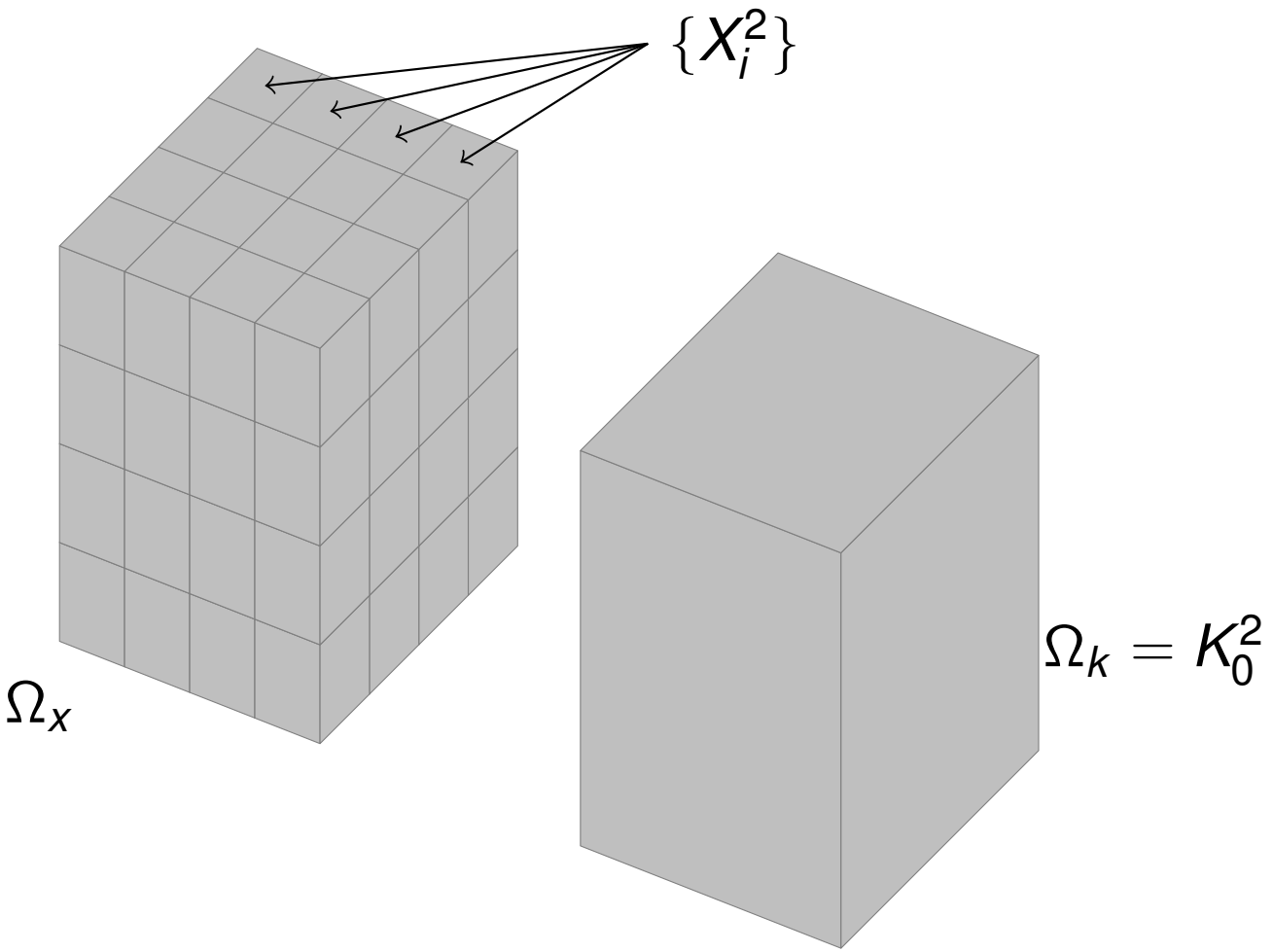
Butterfly Algorithm Example: Level 1

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.



Butterfly Algorithm Example: Level 2

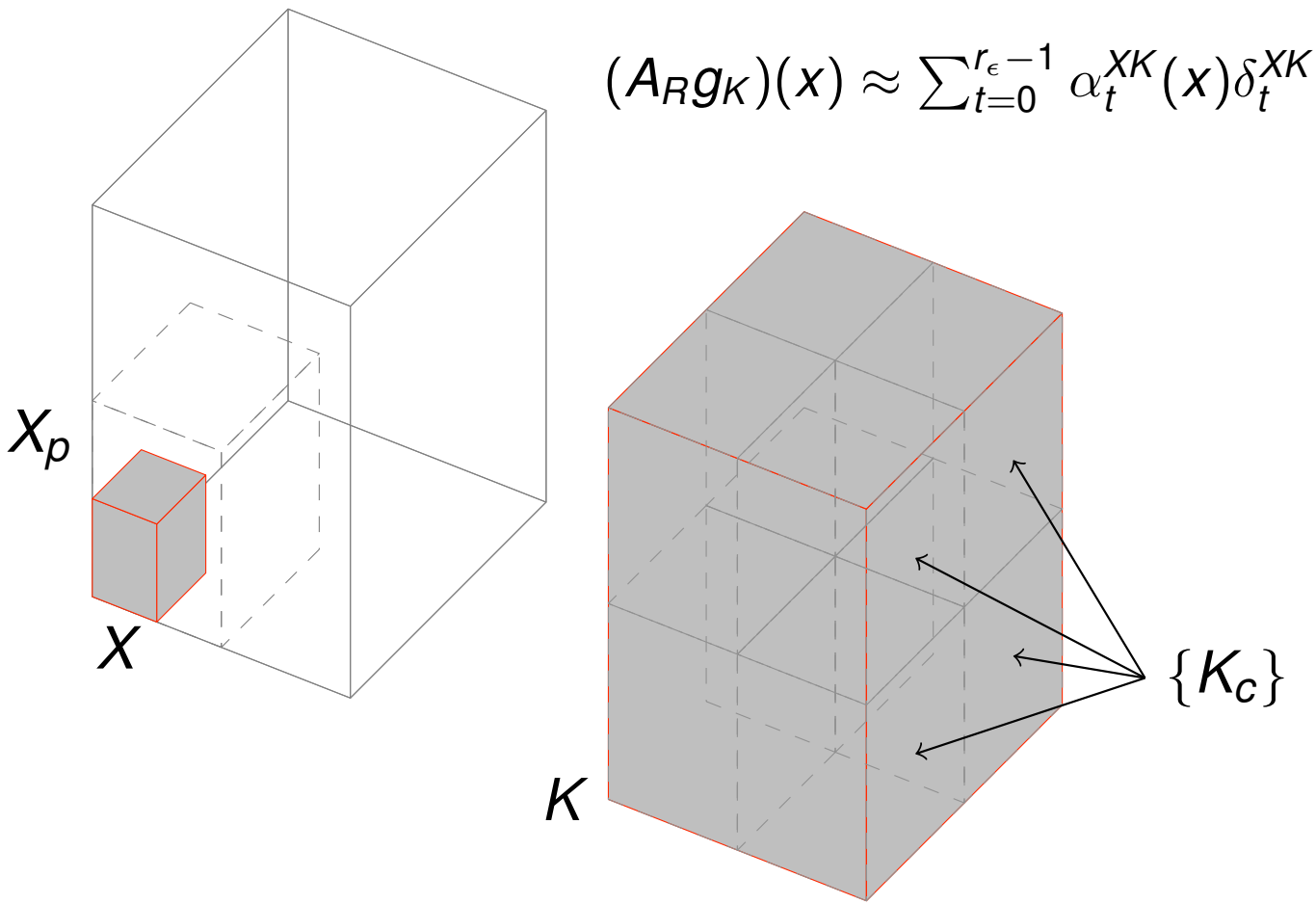
Refine Ω_x and coarsen Ω_k by factor of 2 to form $\{X_i^2\}$ and $\{K_j^2\}$.



Butterfly Algorithm Example: Level 2

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

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



Butterfly Algorithm Example: Summary

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

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

- In general, there are $\log_2(N)$ levels.
- Since $\{X_i^{\log_2(N)}\}$ covers Ω_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

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

- In general, there are $\log_2(N)$ levels.
- Since $\{X_i^{\log_2(N)}\}$ covers Ω_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

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

- In general, there are $\log_2(N)$ levels.
- Since $\{X_i^{\log_2(N)}\}$ covers Ω_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

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

- In general, there are $\log_2(N)$ levels.
- Since $\{X_i^{\log_2(N)}\}$ covers Ω_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

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

- In general, there are $\log_2(N)$ levels.
- Since $\{X_i^{\log_2(N)}\}$ covers Ω_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.

References I


E. Candès, L. Demanet, and L. Ying
A Fast Butterfly Algorithm for the Computation of Fourier Integral Operators
SIAM MMS, 7(4):1727-1750, 2009.


E. Candès, L. Demanet, and L. Ying
Fast Computation of Fourier Integral Operators
SIAM JSC, 29(6):2464-2493, 2007.


E. Michielssen and A. Boag
A multilevel matrix decomposition algorithm for analyzing scattering from large structures.
IEEE AP, 44(8):1086-1093, 1996.

Outline

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 Ω_k fully distributed and end with Ω_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 Ω_k to Ω_x during the coarsening of Ω_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 Ω_k fully distributed and end with Ω_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 Ω_k to Ω_x during the coarsening of Ω_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 Ω_k fully distributed and end with Ω_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 Ω_k to Ω_x during the coarsening of Ω_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 Ω_k fully distributed and end with Ω_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 Ω_k to Ω_x during the coarsening of Ω_k.
- Limit to at most N^d processes (one interaction per process).

Parallelization Strategy

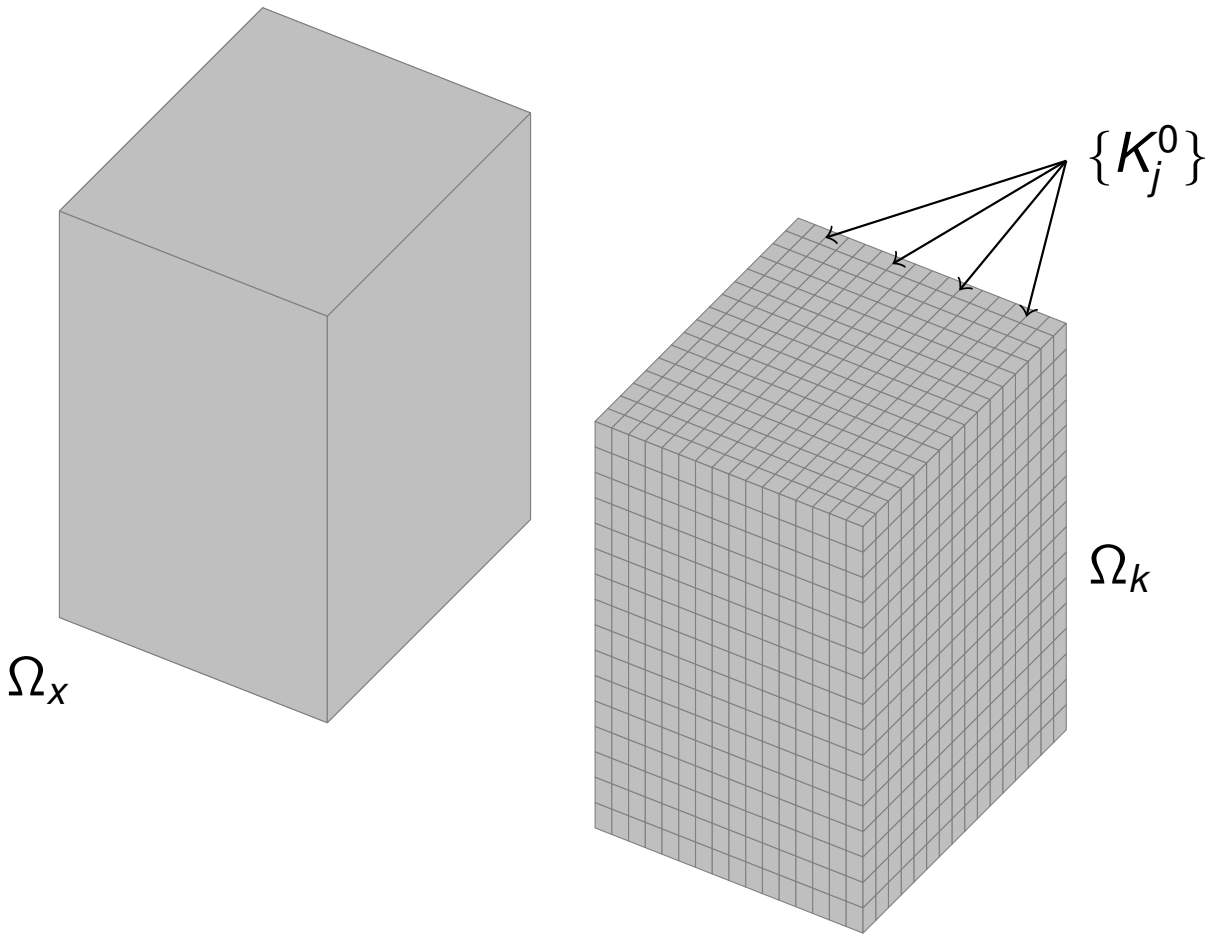
- Grids change at every level, but the number of interactions is constant (N^d).
- The idea is to keep the interactions (and therefore data) evenly distributed.
- Begin with Ω_k fully distributed and end with Ω_x fully distributed.
- Use power of two number of processes so that repeated bisections can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

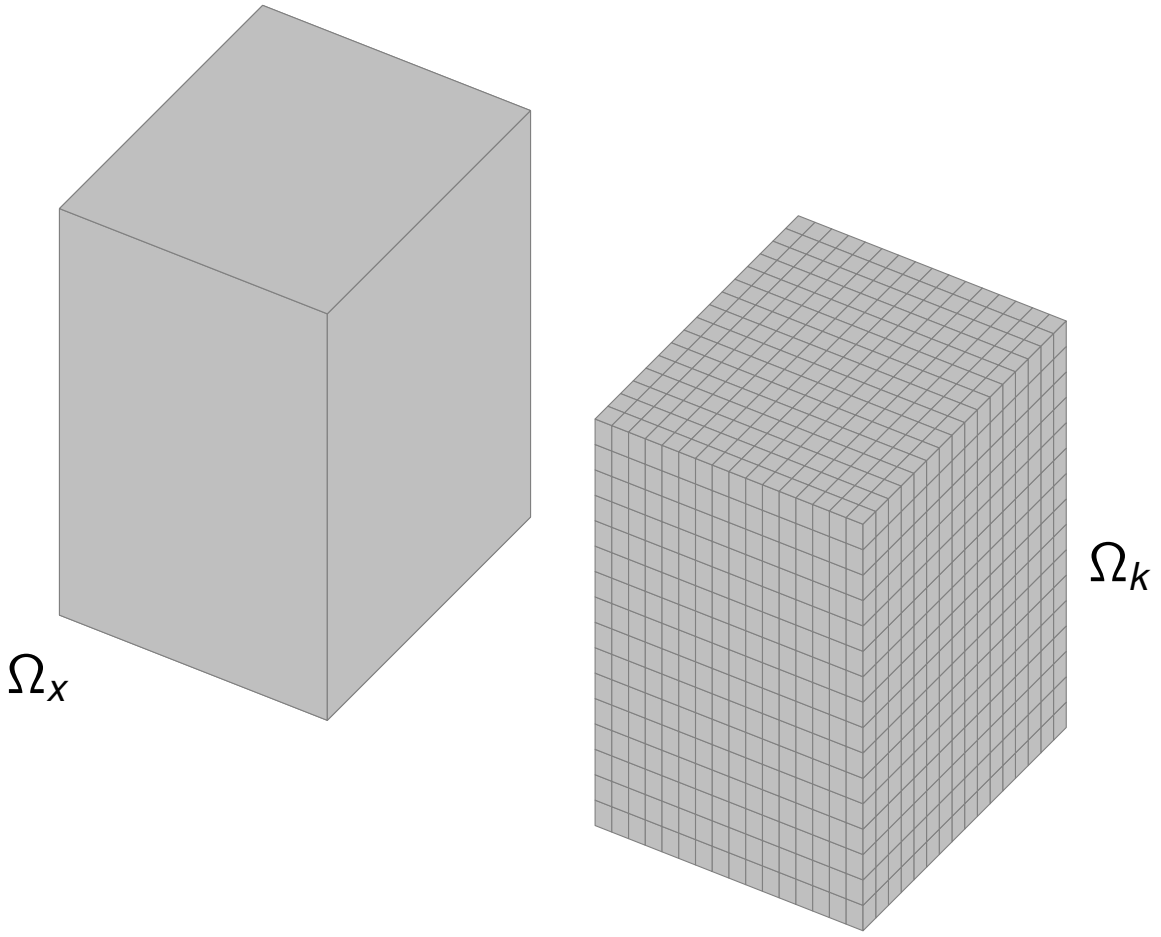
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



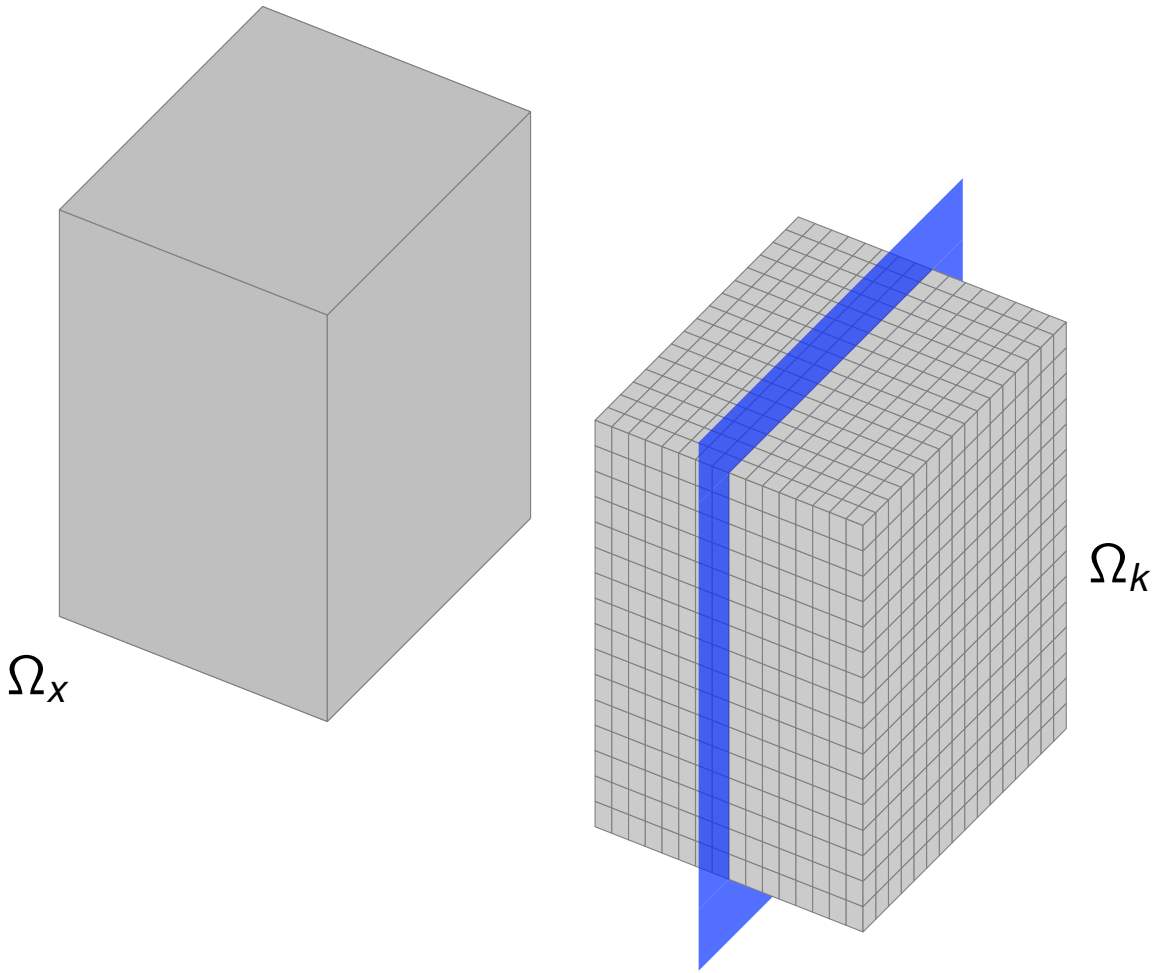
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



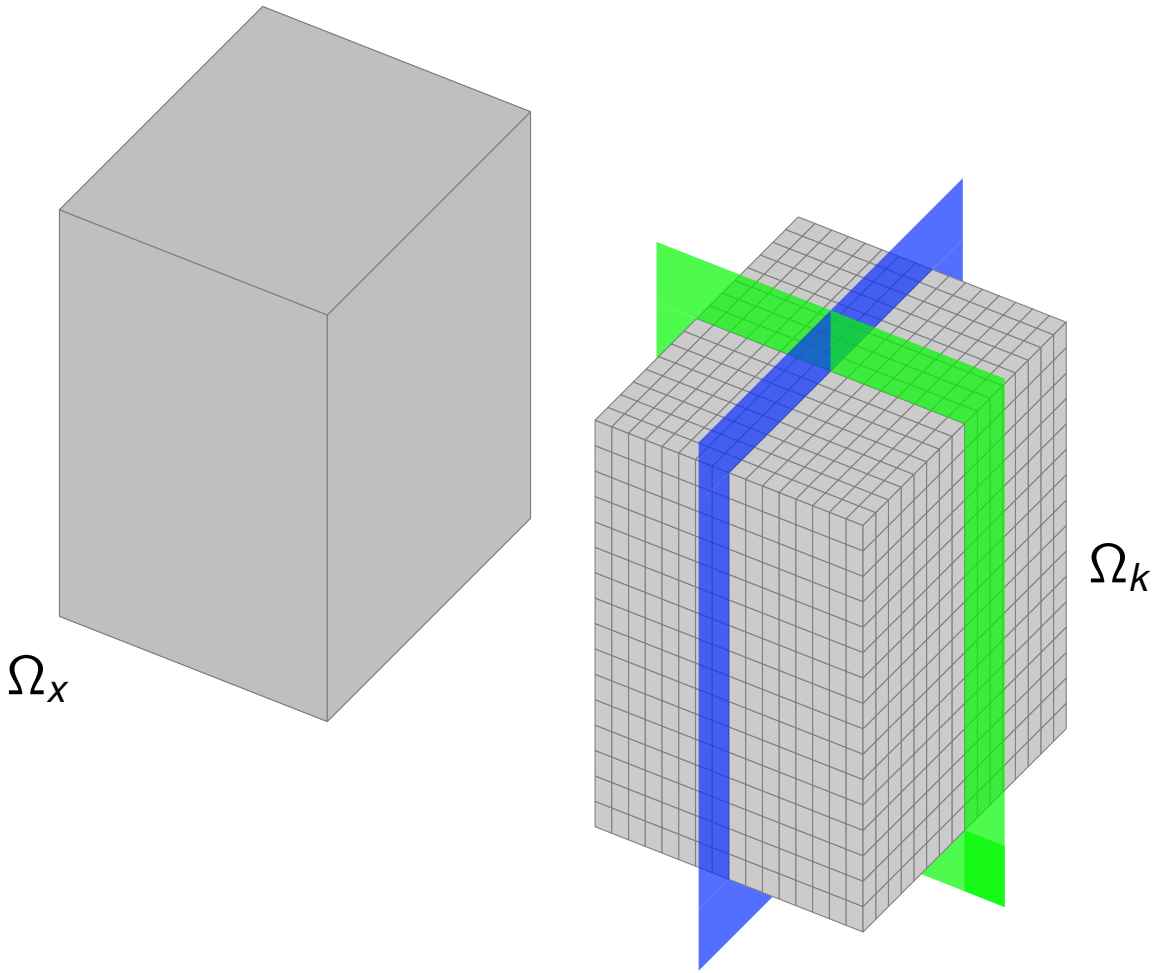
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



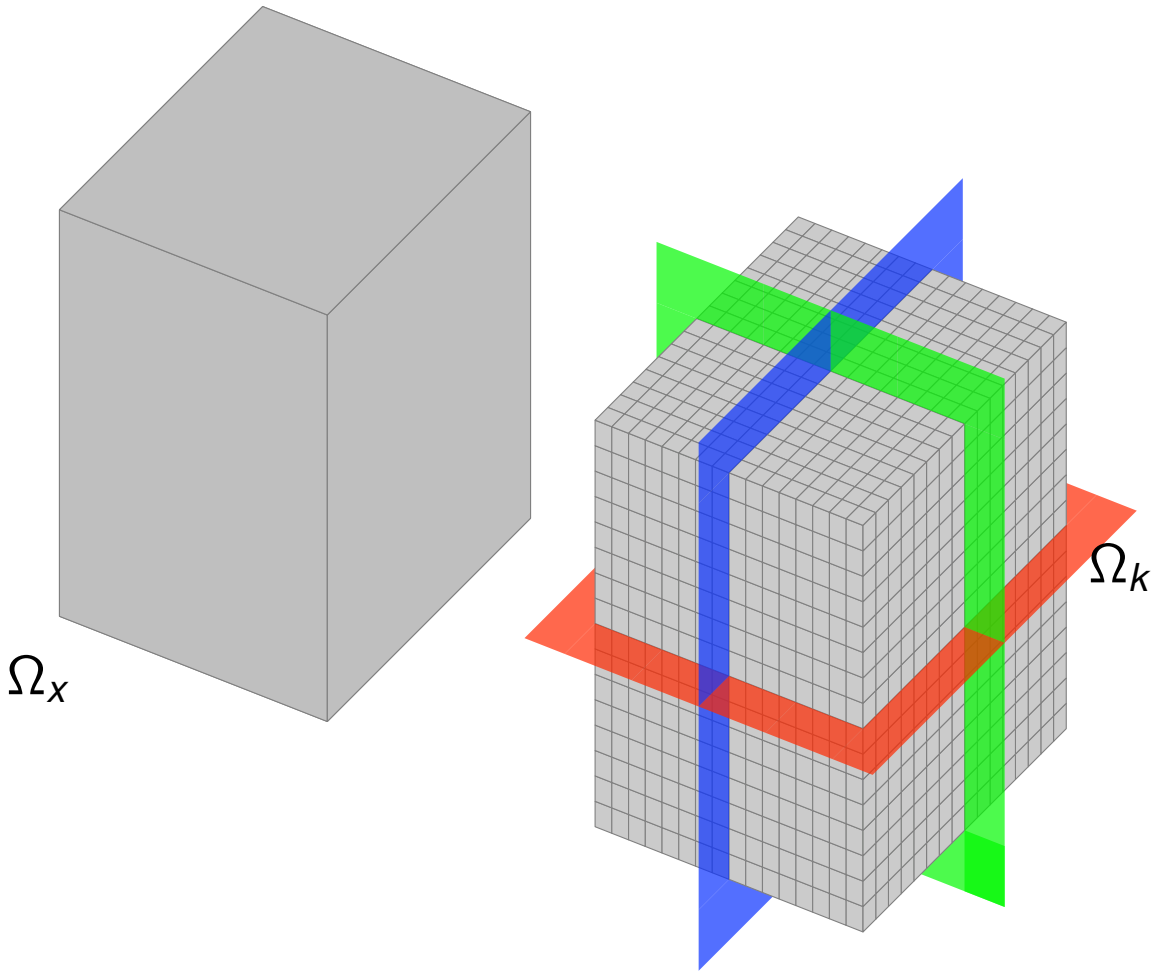
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



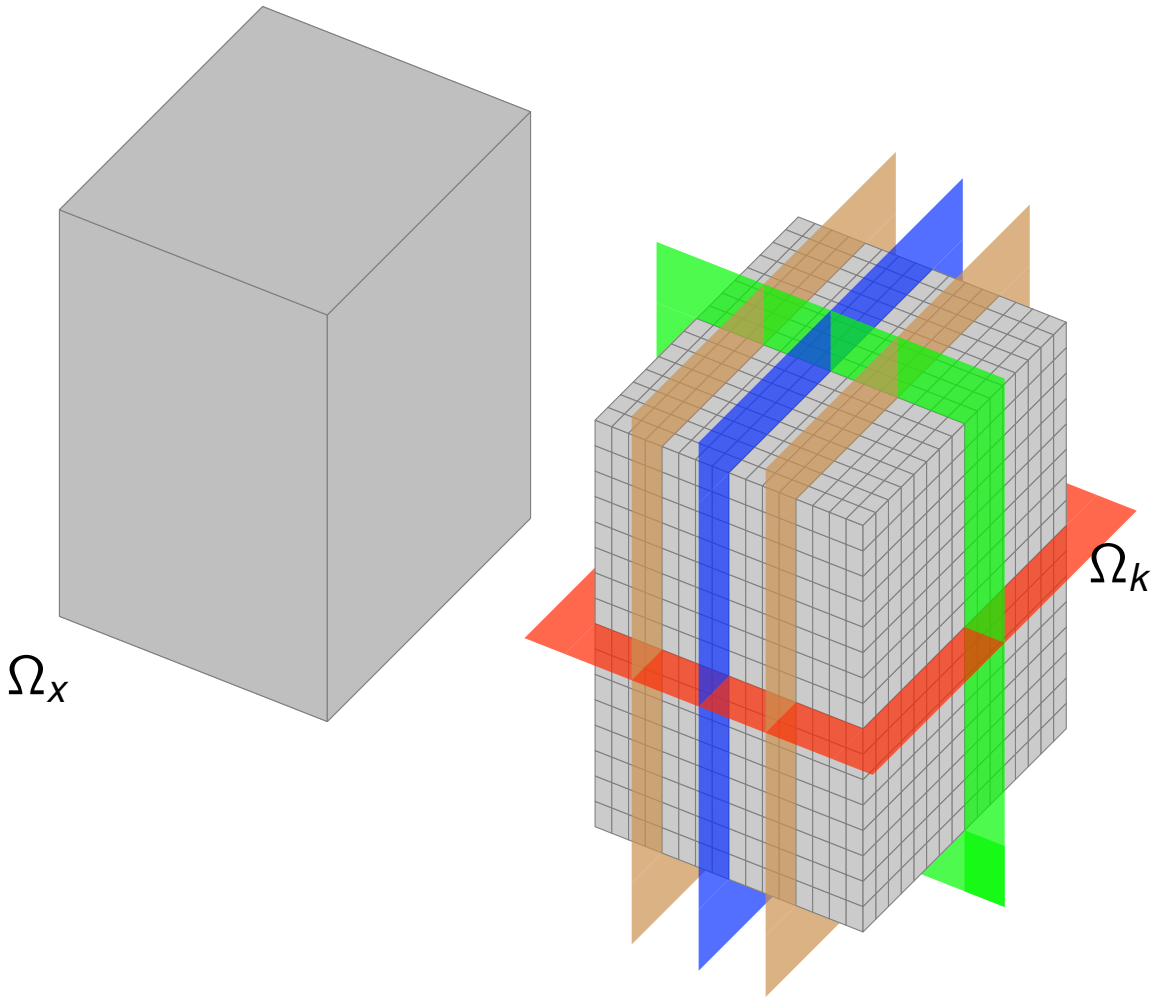
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



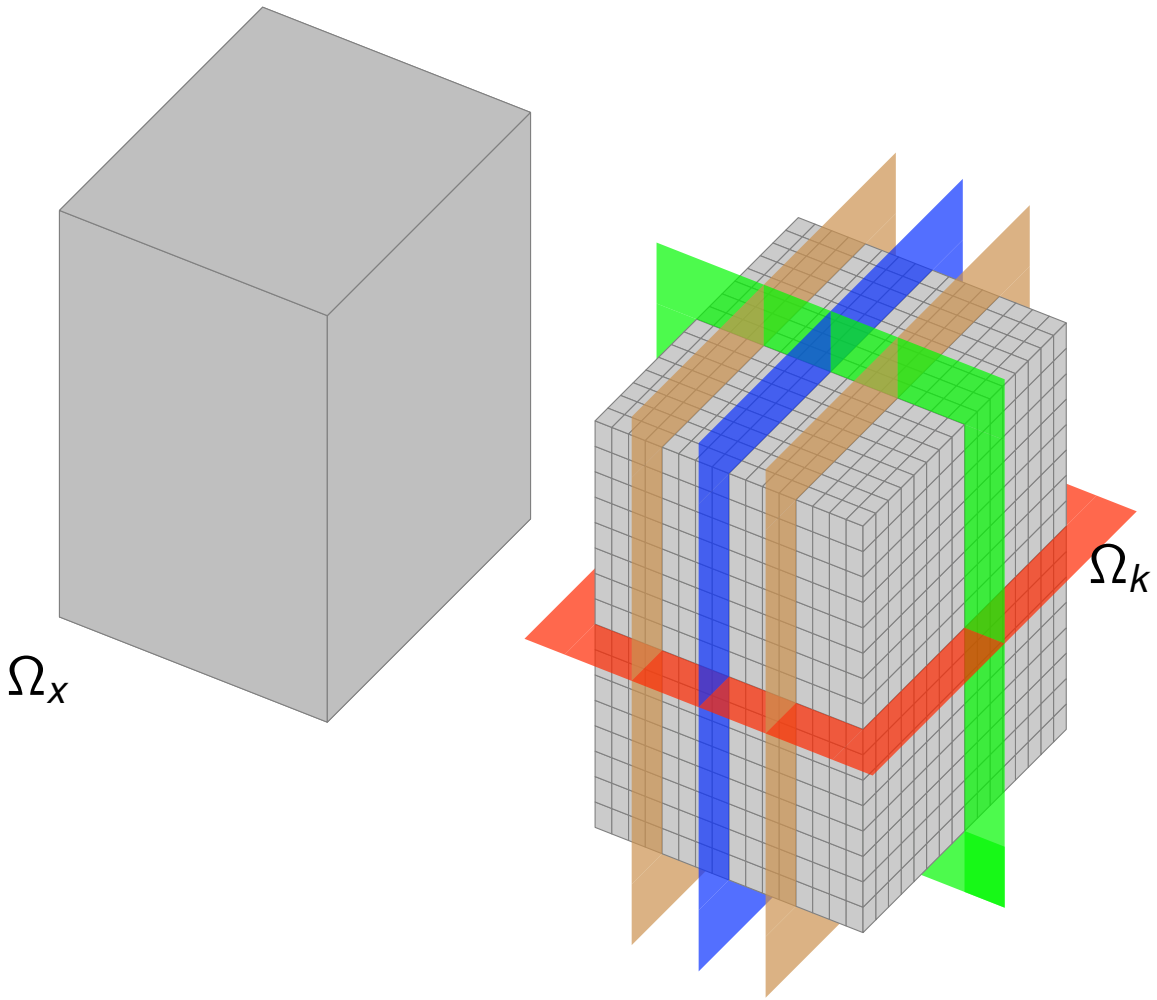
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



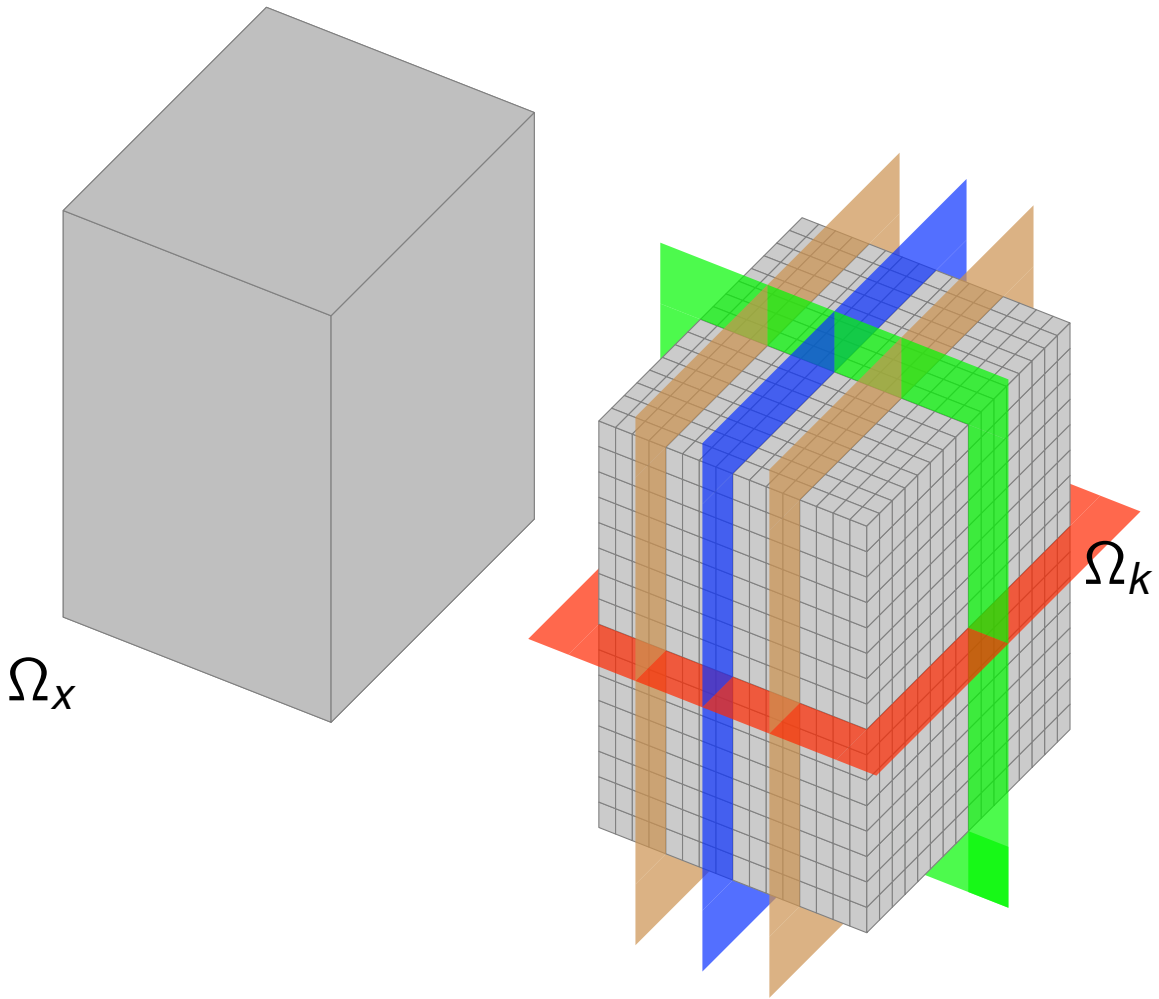
Example for $d = 3$, $N = 16$, and $p = 16$: Level 0

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



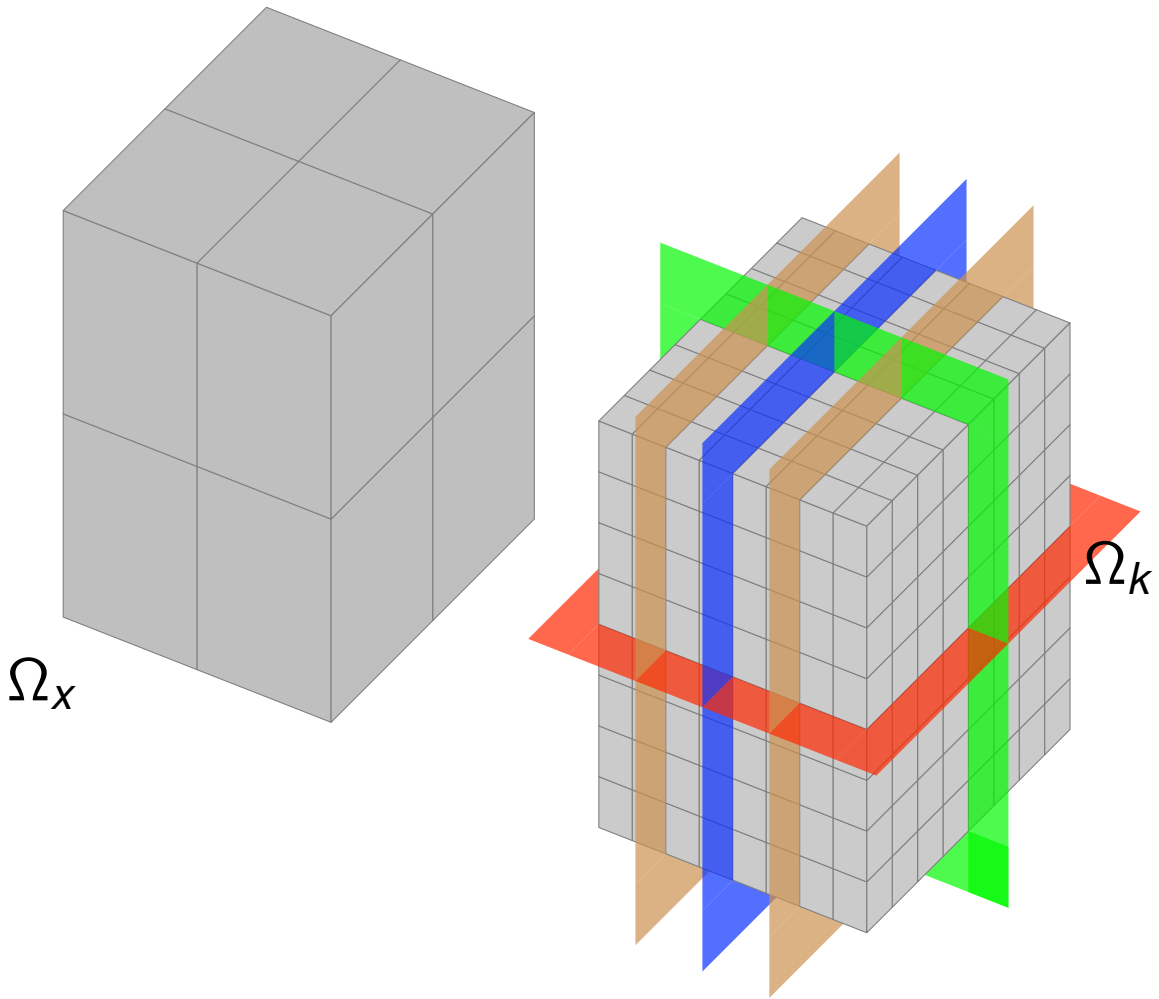
Example for $d = 3$, $N = 16$, and $p = 16$: Level 1

Refinement of Ω_x and coarsening of Ω_k is done locally on each process since no process boundaries are crossed.



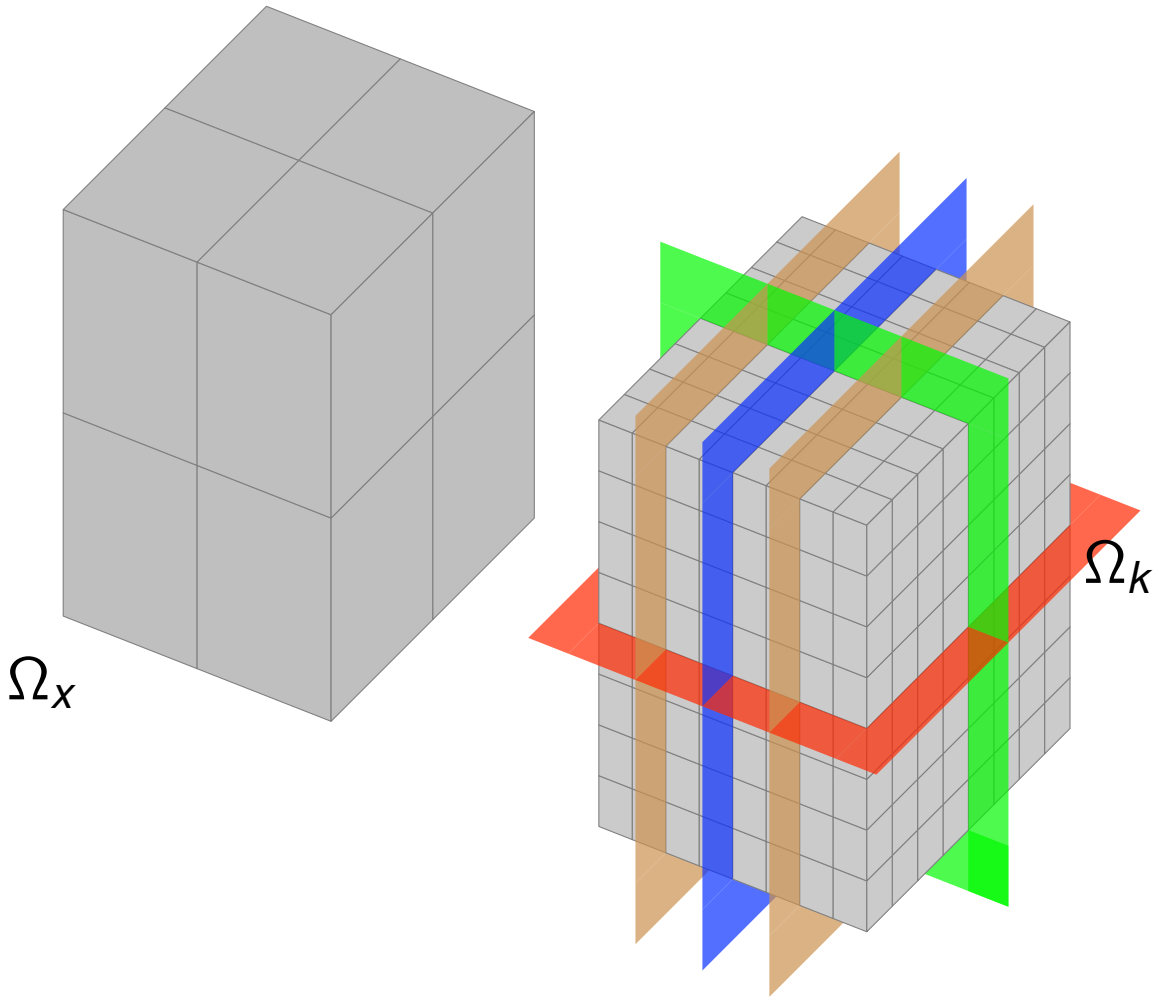
Example for $d = 3$, $N = 16$, and $p = 16$: Level 1

Refinement of Ω_x and coarsening of Ω_k is done locally on each process since no process boundaries are crossed.



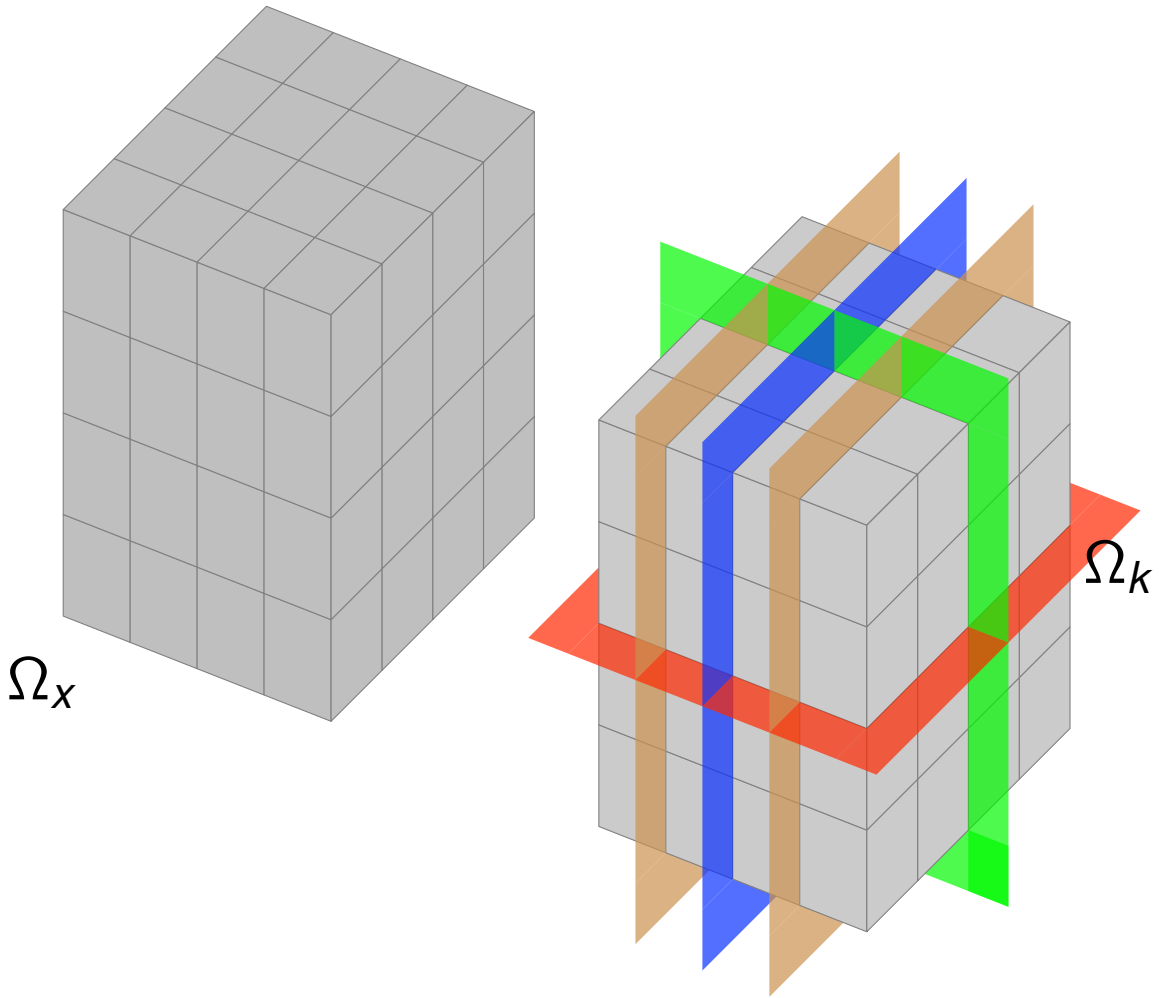
Example for $d = 3$, $N = 16$, and $p = 16$: Level 2

Refinement of Ω_x and coarsening of Ω_k is again local because no process boundaries are crossed.



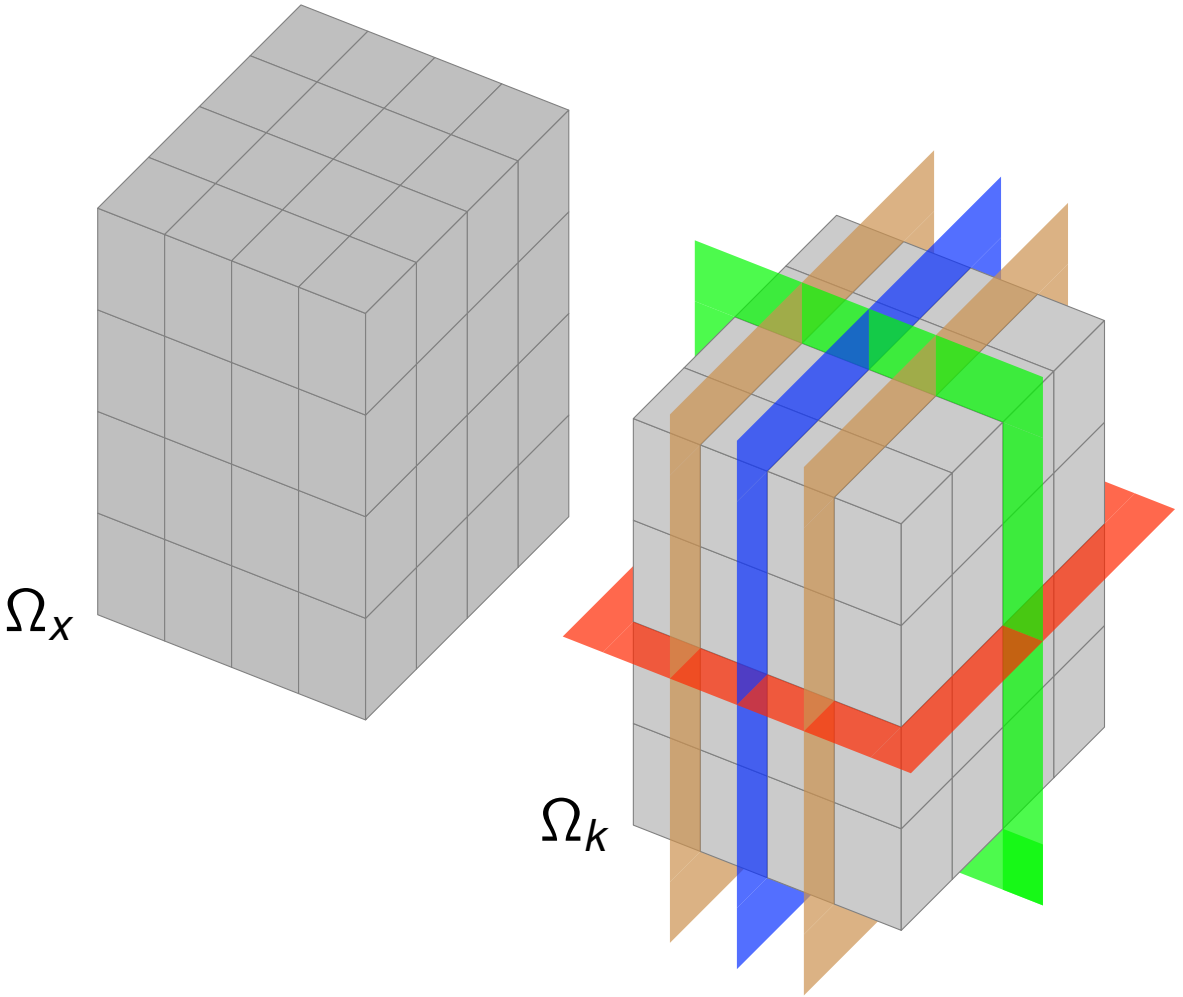
Example for $d = 3$, $N = 16$, and $p = 16$: Level 2

Refinement of Ω_x and coarsening of Ω_k is again local because no process boundaries are crossed.



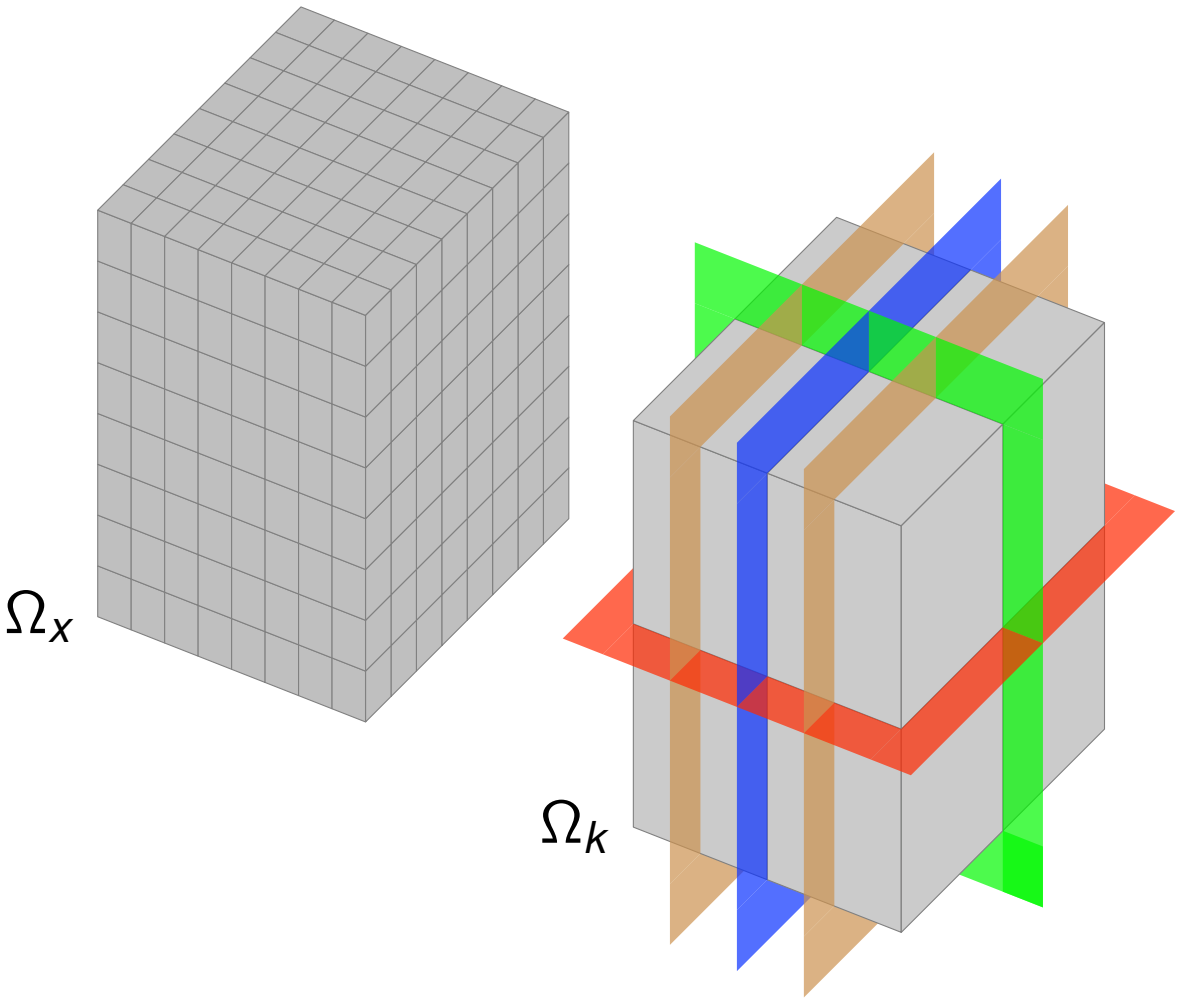
Example for $d = 3$, $N = 16$, and $p = 16$: Level 3

Coarsening Ω_k crosses the fourth set of bisections (in brown).



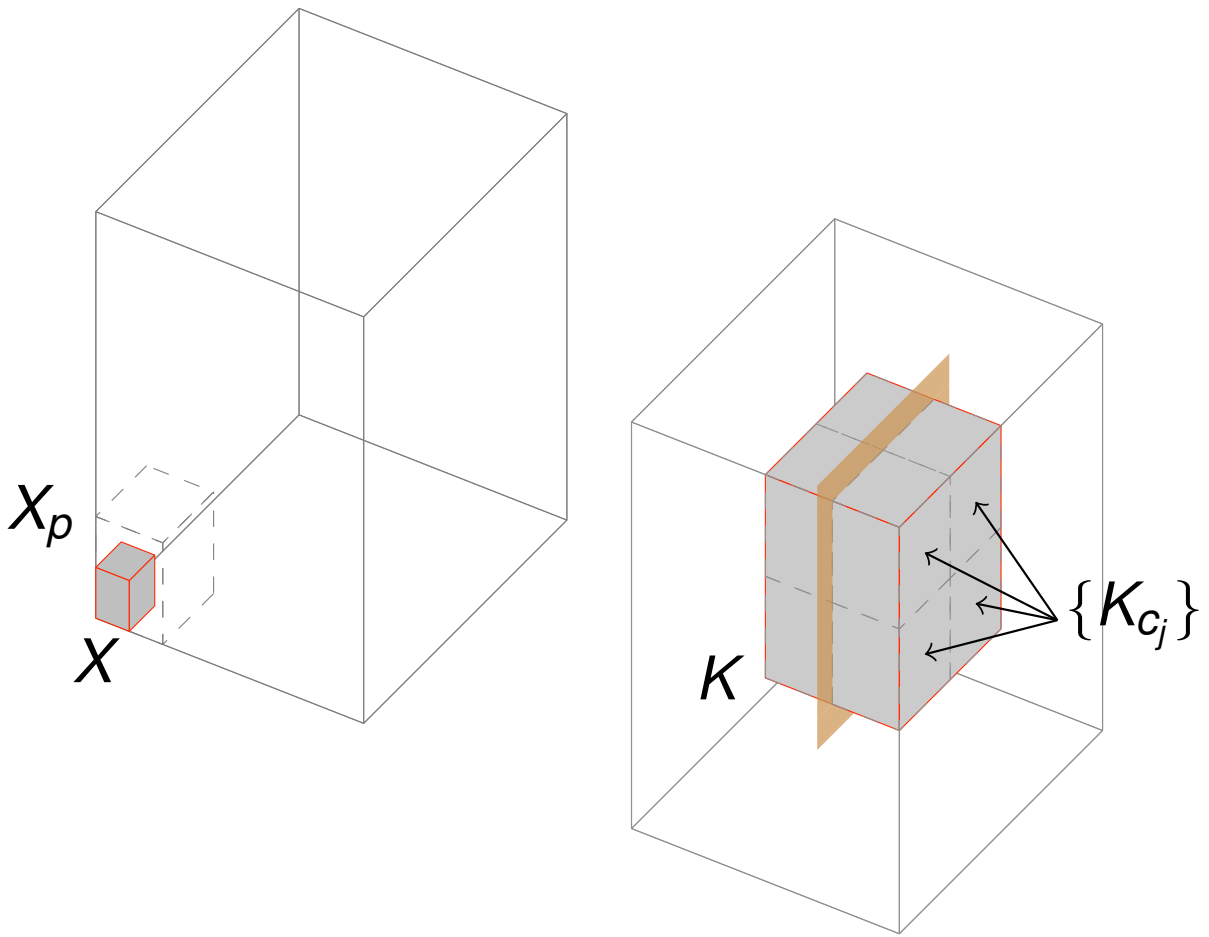
Example for $d = 3$, $N = 16$, and $p = 16$: Level 3

Coarsening Ω_k crosses the fourth set of bisections (in brown).



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

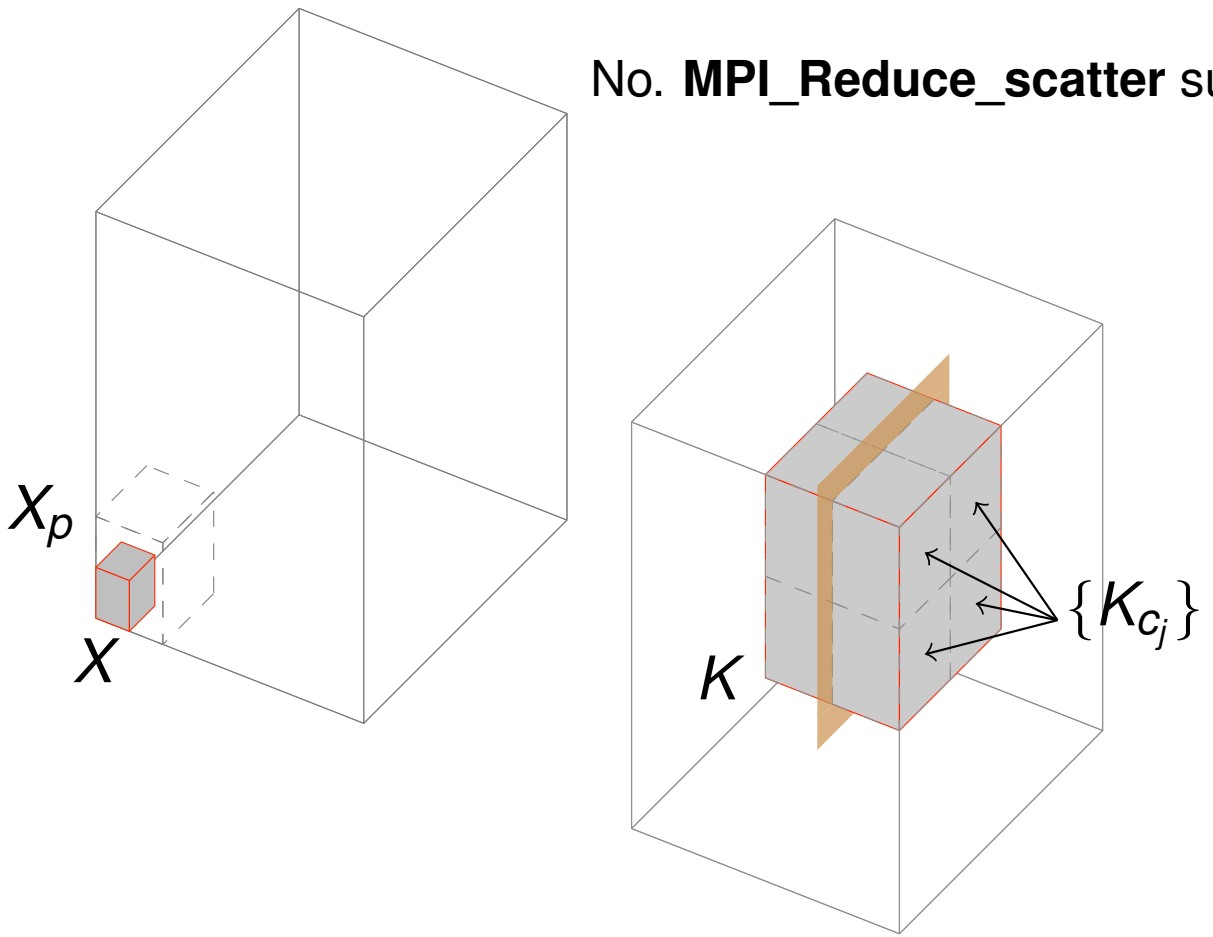
The $\{\delta_t^{XK}\}$ weights are a sum of the contributions from each (X_p, K_{c_j}). **MPI_Reduce** sums in teams of 2^m, m crossed parts.?



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

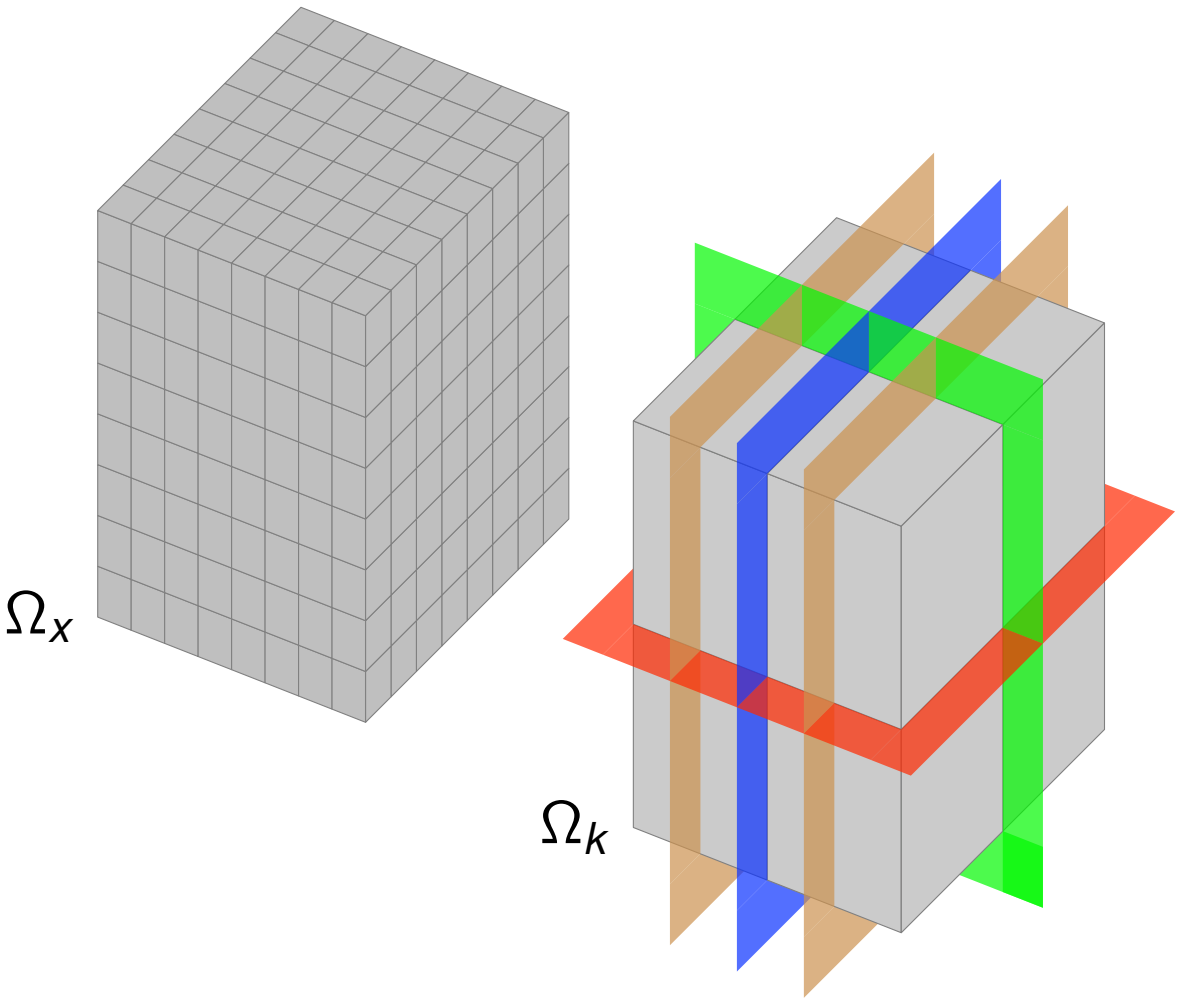
The $\{\delta_t^{XK}\}$ weights are a sum of the contributions from each (X_p, K_{c_j}). **MPI_Reduce** sums in teams of 2^m, m crossed parts.?

No. **MPI_Reduce_scatter** sum in teams of 2^m.



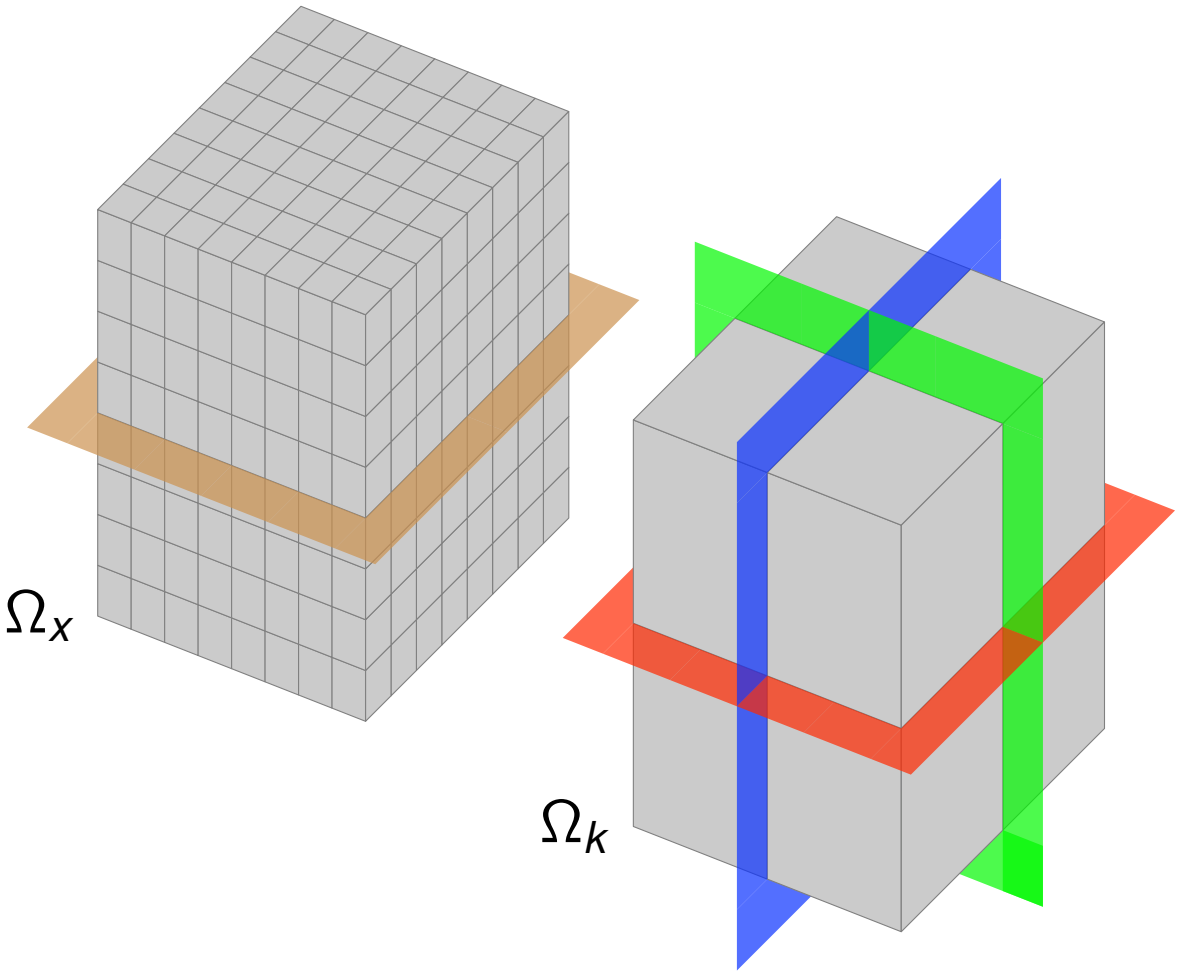
Example for $d = 3$, $N = 16$, and $p = 16$: Level 3

The **MPI_Reduce_scatter** is performed so that it effectively moves the crossed Ω_k partitions onto Ω_x.



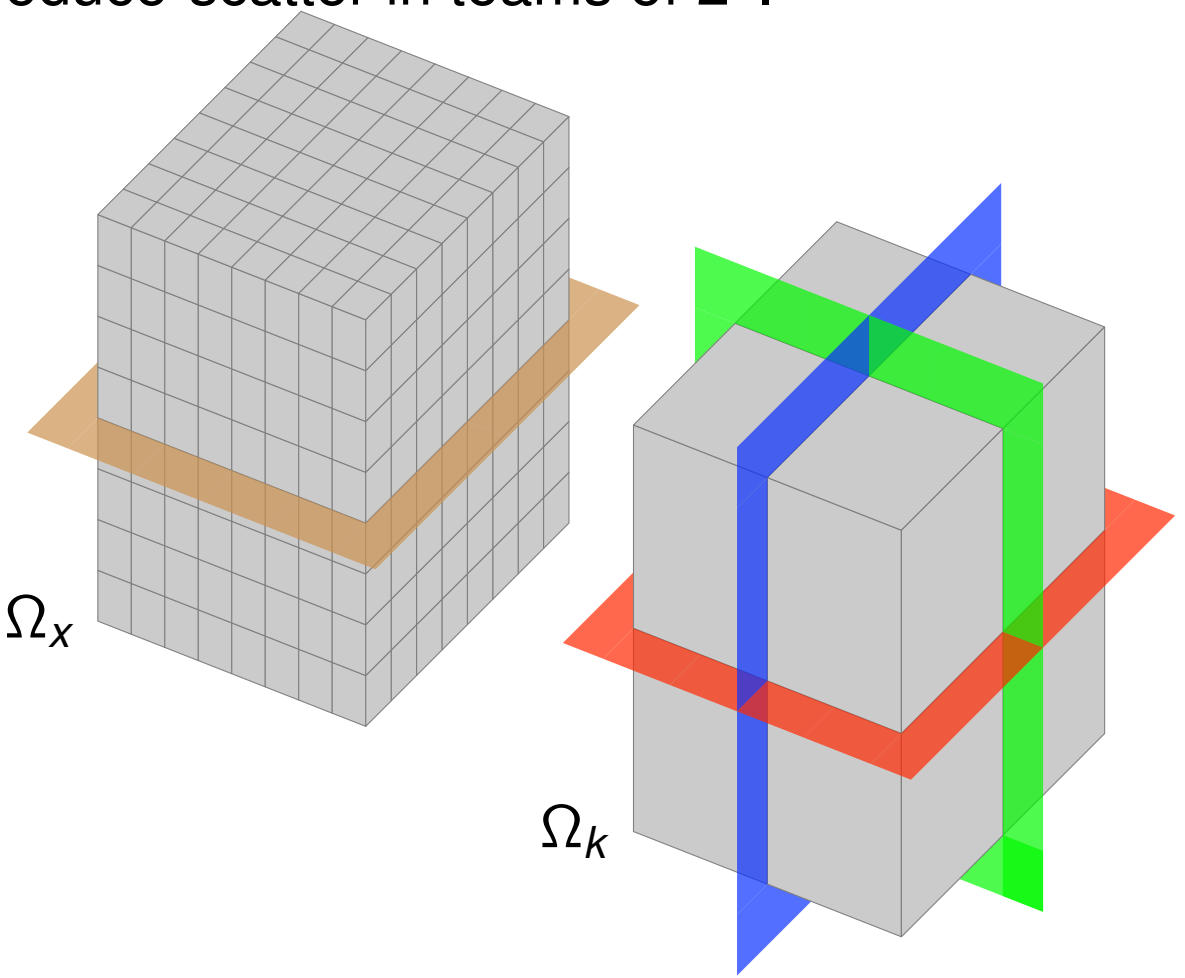
Example for $d = 3$, $N = 16$, and $p = 16$: Level 3

The **MPI_Reduce_scatter** is performed so that it effectively moves the crossed Ω_k partitions onto Ω_x.



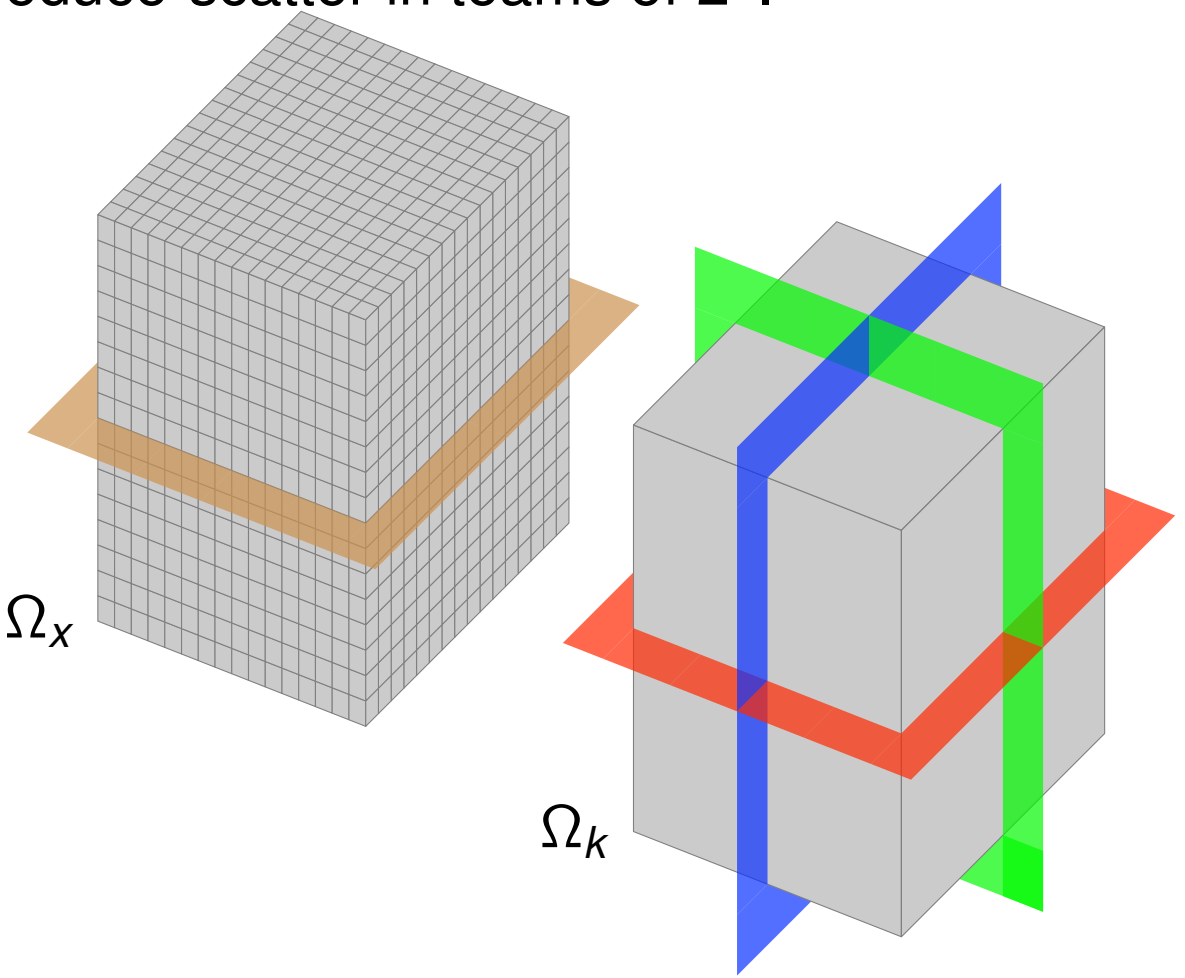
Example for $d = 3$, $N = 16$, and $p = 16$: Level 4

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



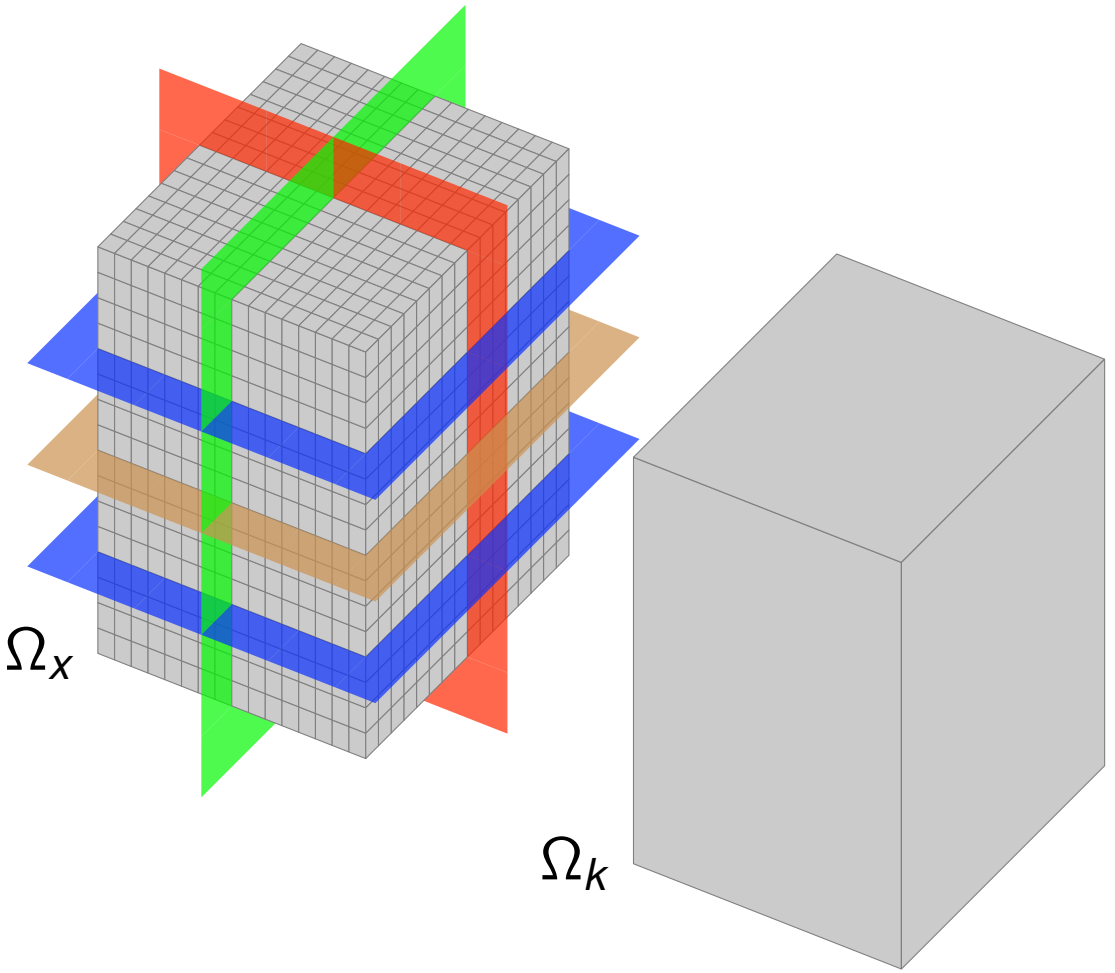
Example for $d = 3$, $N = 16$, and $p = 16$: Level 4

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



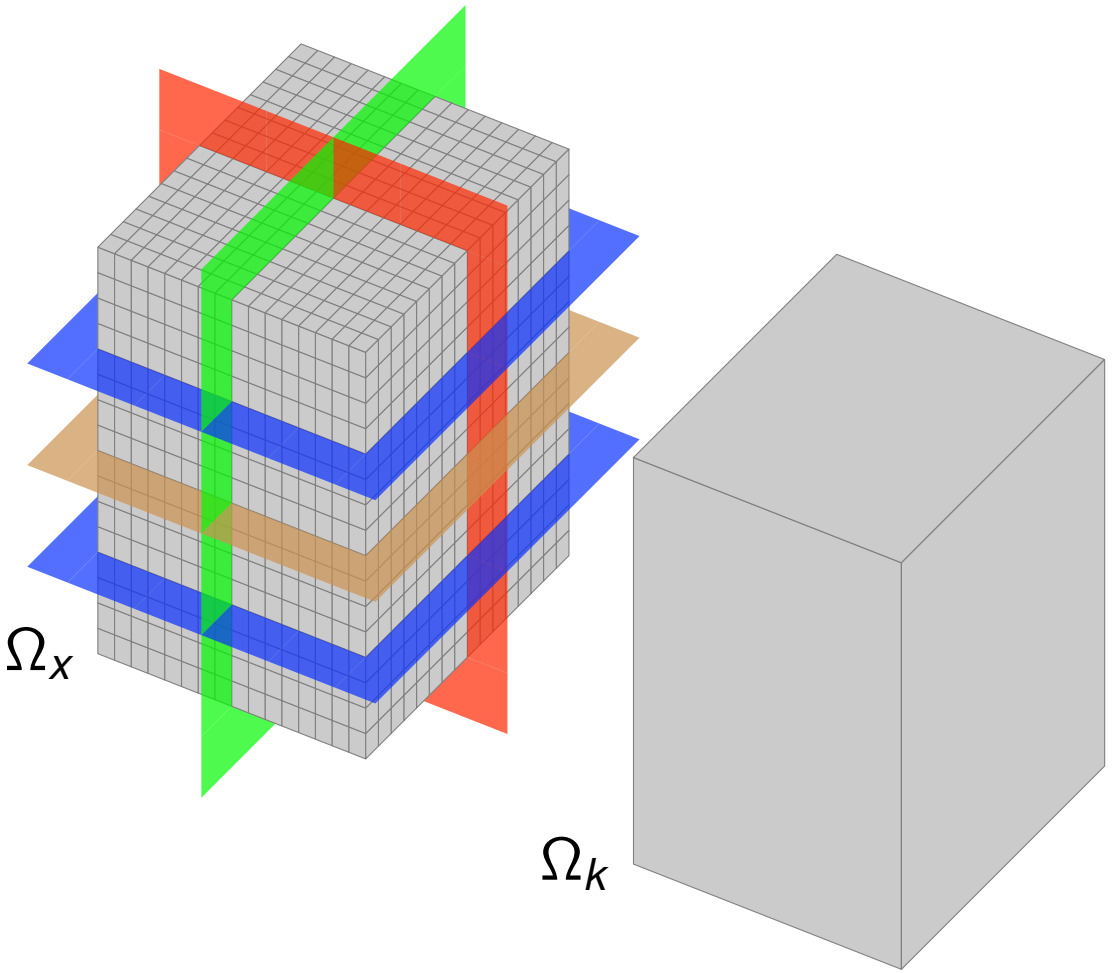
Example for $d = 3$, $N = 16$, and $p = 16$: Level 4

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

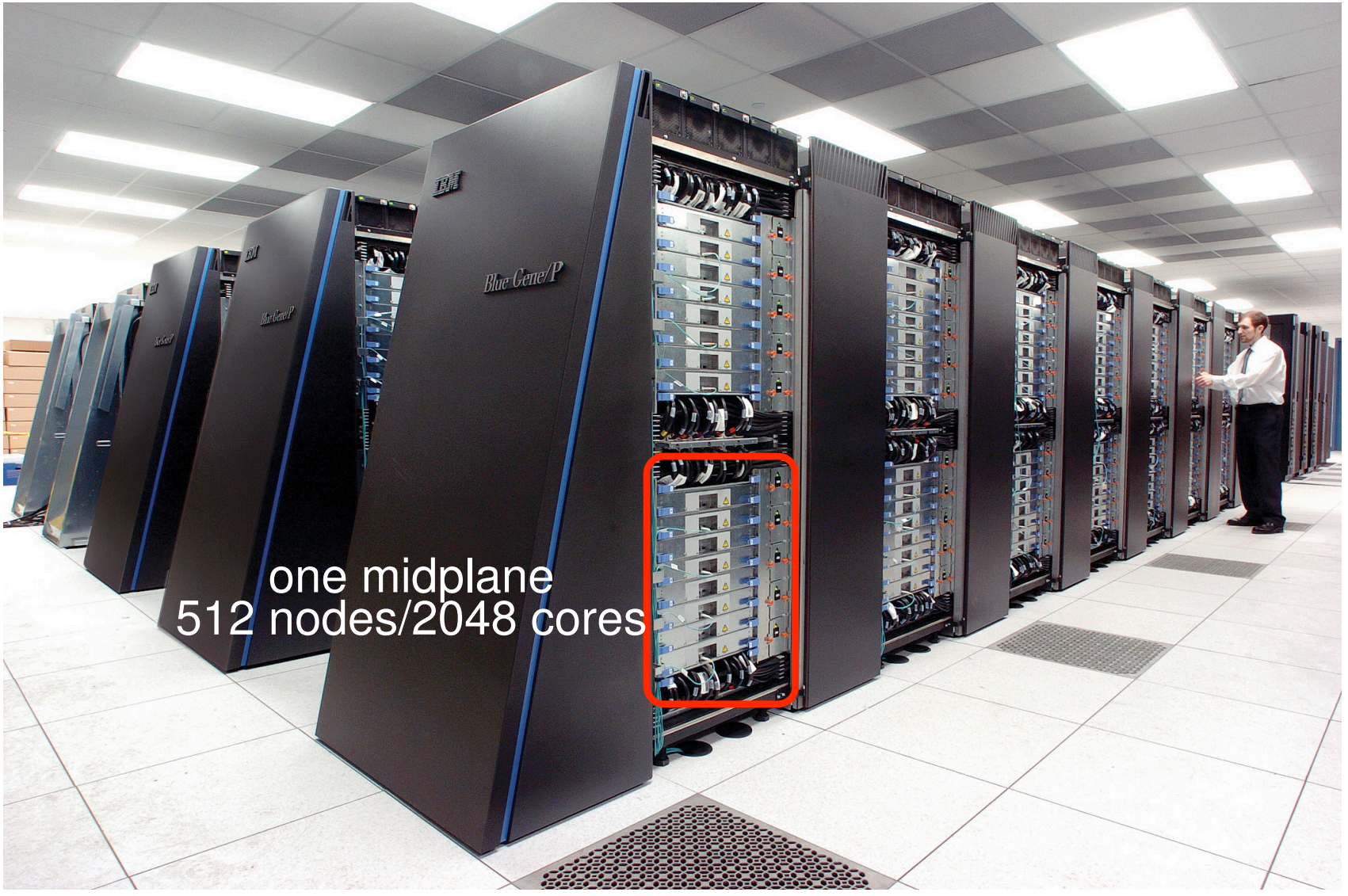
Done! Each process can now locally evaluate $(A_R g)(x)$ for all x within its chunk of Ω_x.



Outline

Blue Gene/P

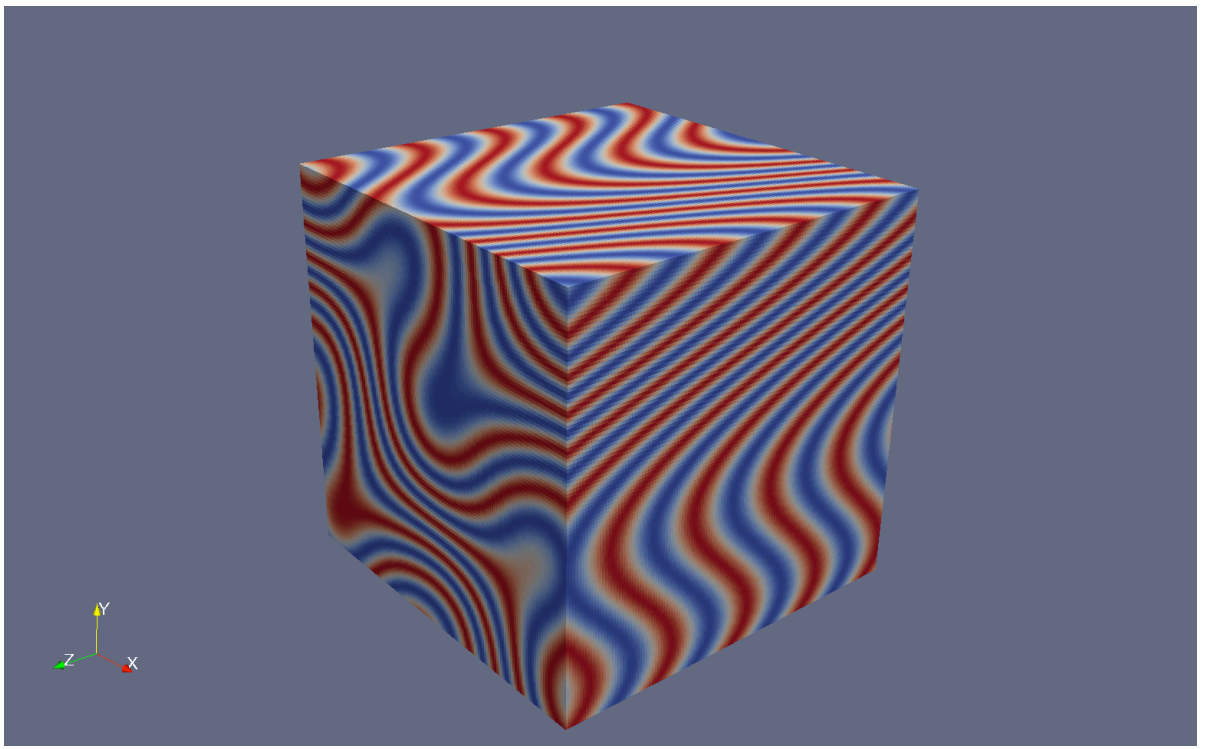
Blue Gene/P



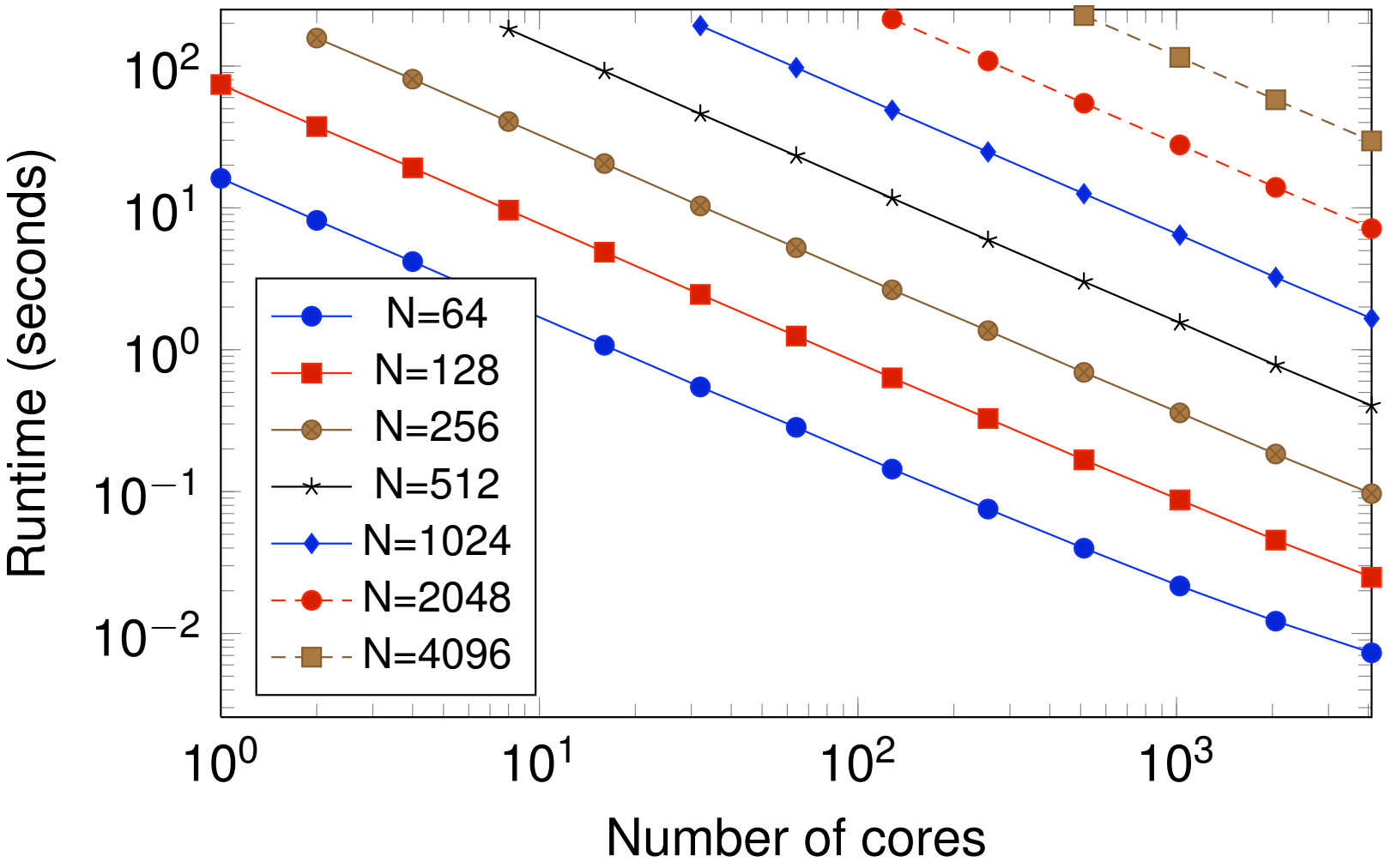
Blue Gene/P

Analogue of Generalized Radon Transform

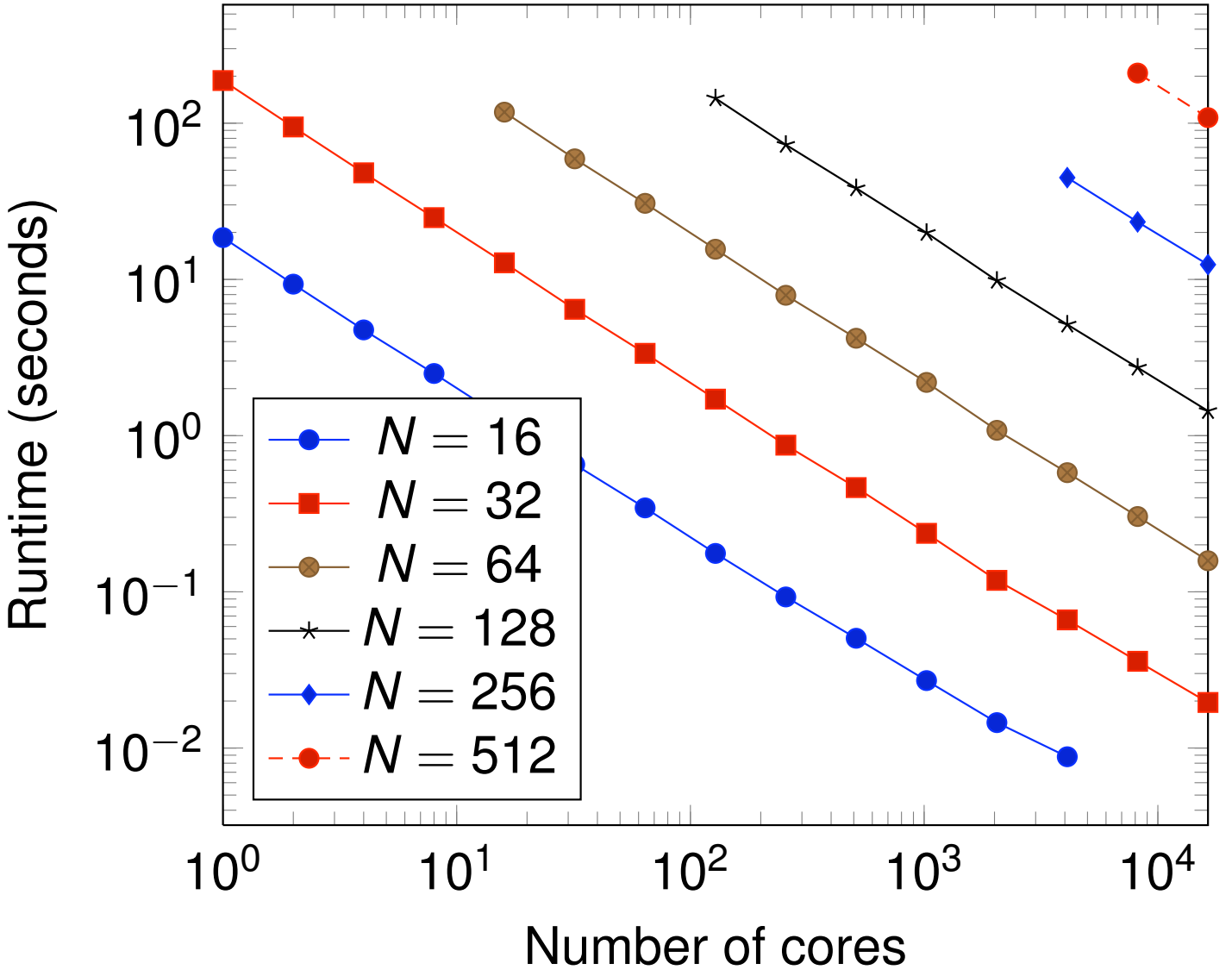
$$\begin{aligned} \alpha &= k_0(2+\sin(2\pi x_0)\sin(2\pi x_1))/3 \\ \beta &= k_1(2+\cos(2\pi x_0)\cos(2\pi x_1))/3 \\ \phi(x,k) &= 2\pi x\cdot k+\sqrt{\alpha^2+\beta^2}. \end{aligned}$$



Non-uniform 2d Generalized Radon Analogue, $q = 8$



Non-uniform 3d Fourier Transform, $q = 5$



Outline

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!

Defining your phase functor

```
1  // Create a 3d single-precision phase functor
2  class MyPhase : public bfio::Phase<float,3> {
3  public:
4      // Must define a function that returns the evaluation at (x,k)
5      virtual float operator()
6      ( const bfio::Array<float,3>& x,
7        const bfio::Array<float,3>& k ) const
8      { return x[0]*k[0]+x[1]*k[1]+x[2]*k[2] +
9               sqrt(k[0]*k[0]+k[1]*k[1]); }
10
11     // Optional, can improve performance
12     virtual void BatchEvaluate
13     ( const std::vector< bfio::Array<float,3> >& xPoints,
14       const std::vector< bfio::Array<float,3> >& kPoints,
15             std::vector< float                >& results ) const
16     { /* Define here. */ }
17
18     // Unfortunately we need a 'virtual constructor'
19     virtual MyPhase* Clone() const
20     { return new MyPhase(*this); }
21 };
```

Defining the domains

```
1  // Initialize coefficients for 3-dimensional problems with q-th order
2  bfio::rfio::Context<float,3,q> context;
3
4  // Choose the size of our problem
5  const std::size_t N = 64;
6
7  // Define the source and target boxes
8  bfio::Box<float,3> sourceBox, targetBox;
9  for( std::size_t j=0; j<3; ++j ) {
10     sourceBox.offsets[j] = -0.5*N; sourceBox.widths[j] = N;
11     targetBox.offsets[j] = 0;      targetBox.widths[j] = 1;
12 }
13
14 // Initialize parallel strategy
15 bfio::Plan<d> plan( MPI_COMM_WORLD, bfio::FORWARD, N );
16
17 // Extract this process's portion of the source domain
18 bfio::Box<float,3> mySourceBox =
19   plan.GetMyInitialSourceBox( sourceBox );
```

Running the transform

```
1  // Create our list of sources.
2  std::vector< bfio::Source<float,3> > mySources;
3  // FILL THIS PROCESS'S SOURCE LOCATIONS AND MAGNITUDES HERE...
4
5  // Create a smart pointer for our portion of the solution
6  std::auto_ptr< const bfio::rfio::PotentialField<float,3,q> > u;
7
8  // Execute the transformation
9  u = bfio::ReducedFIO
10 ( context, plan, phaseFunctor, sourceBox, targetBox, mySources );
11
12 // Evaluate the potential somewhere in our portion of the target
13 const bfio::Box<float,3>& myTargetBox = u.GetMyTargetBox();
14 bfio::Array<float,3> x;
15 for( std::size_t j=0; j<3; ++j )
16     x[j] = myTargetBox.offsets[j] + 0.5*myTargetBox.widths[j];
17 std::complex<float> approx = u.Evaluate( x );
18
19 // Create parallel visualization files
20 bfio::rfio::WriteVtkXmlPImageData
21 ( MPI_COMM_WORLD, N, targetBox, *u, "filename" );
```

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