Distributed Parallelism with HPX (2)

Lecture 20

Hartmut Kaiser

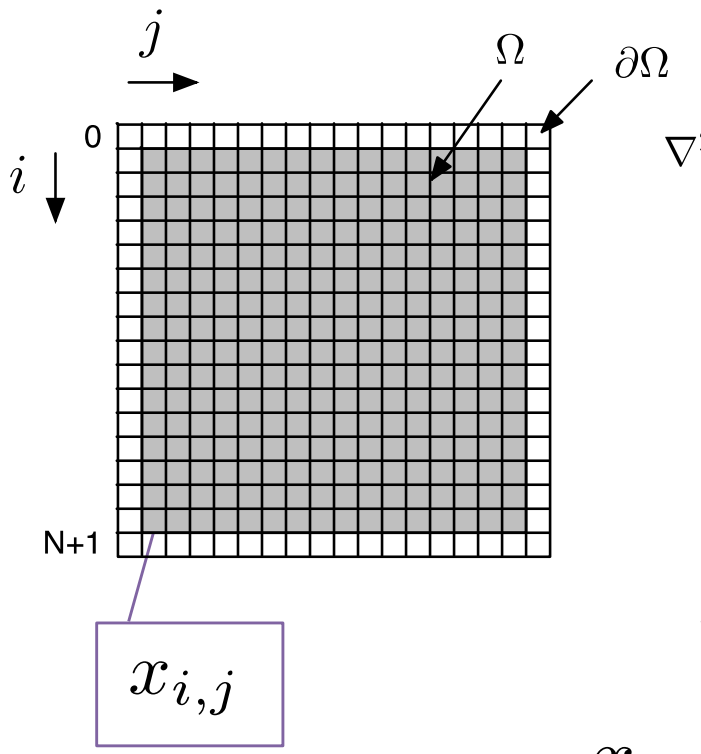
https://teaching.hkaiser.org/spring2025/csc4400/

Overview

- SPMD / CSP recap
- A simple mental model
- Basic HPX
- Four Function HPX Point to Point Version
- Four Function HPX Collective Version
- Laplace's equation on a regular grid

A Distributed Laplace's Equation Solver

Laplace's Equation on a Regular Grid



$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

Discretization

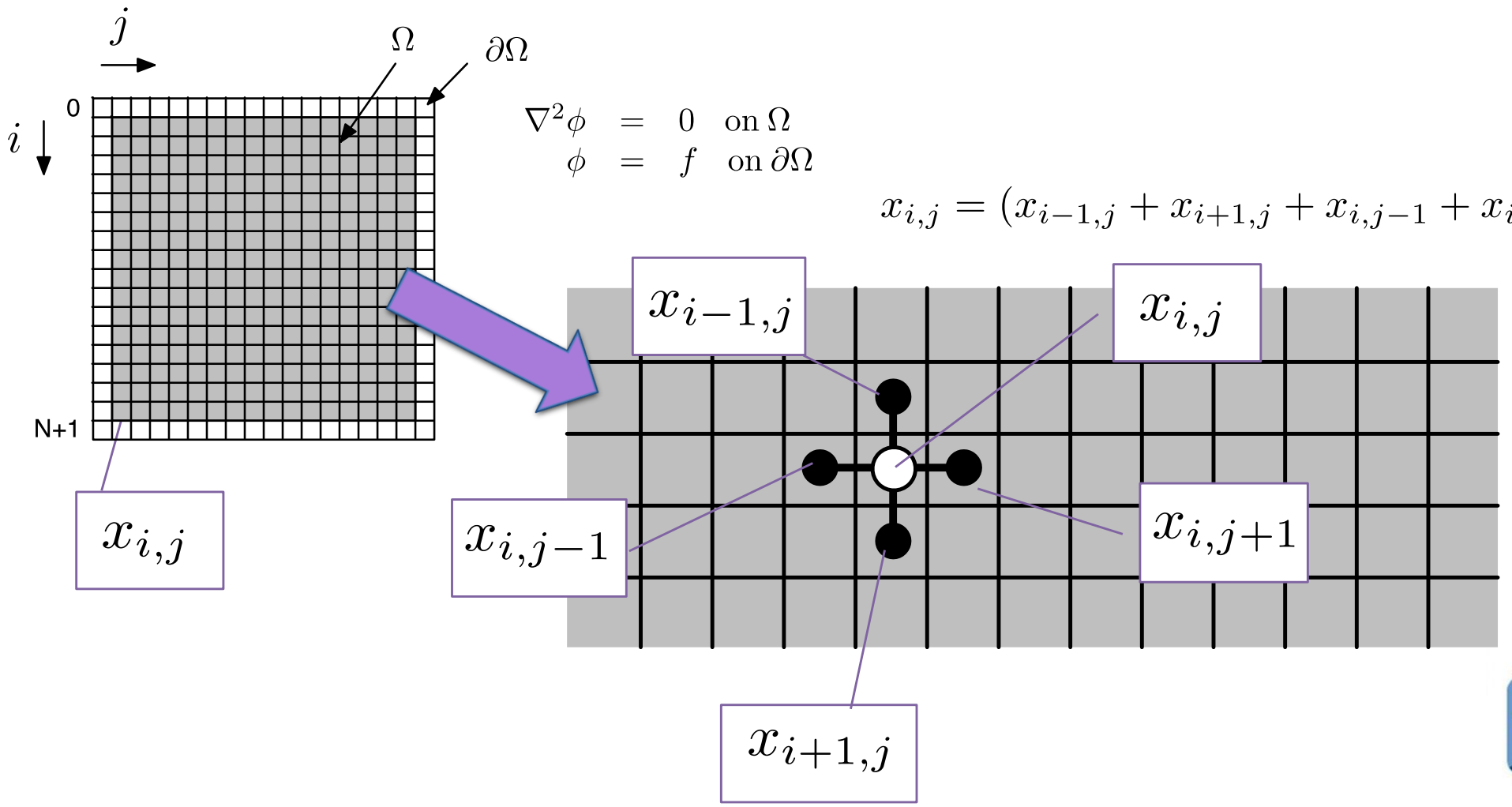
$$x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j} = 0$$

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

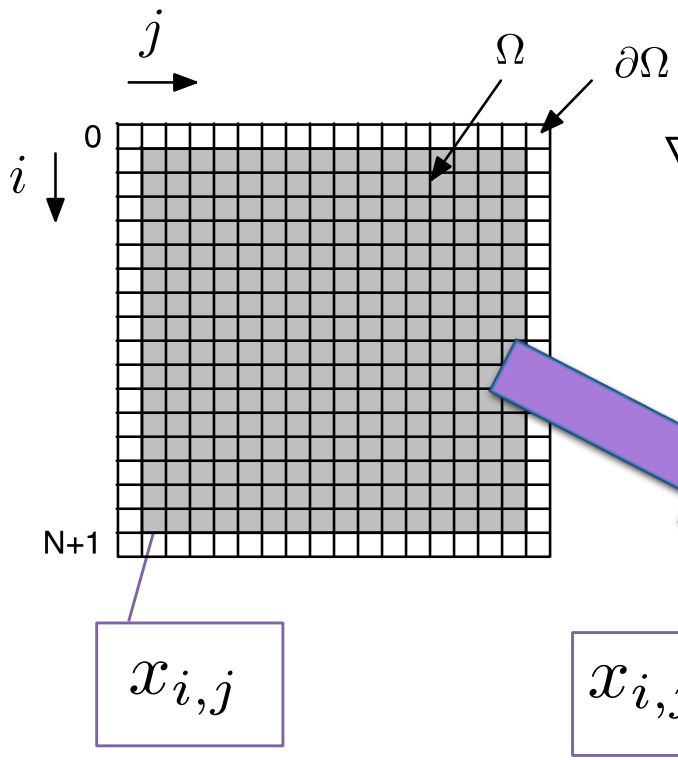
The values of each point on the grid

The average of its neighbors

Laplace's Equation on a Regular Grid



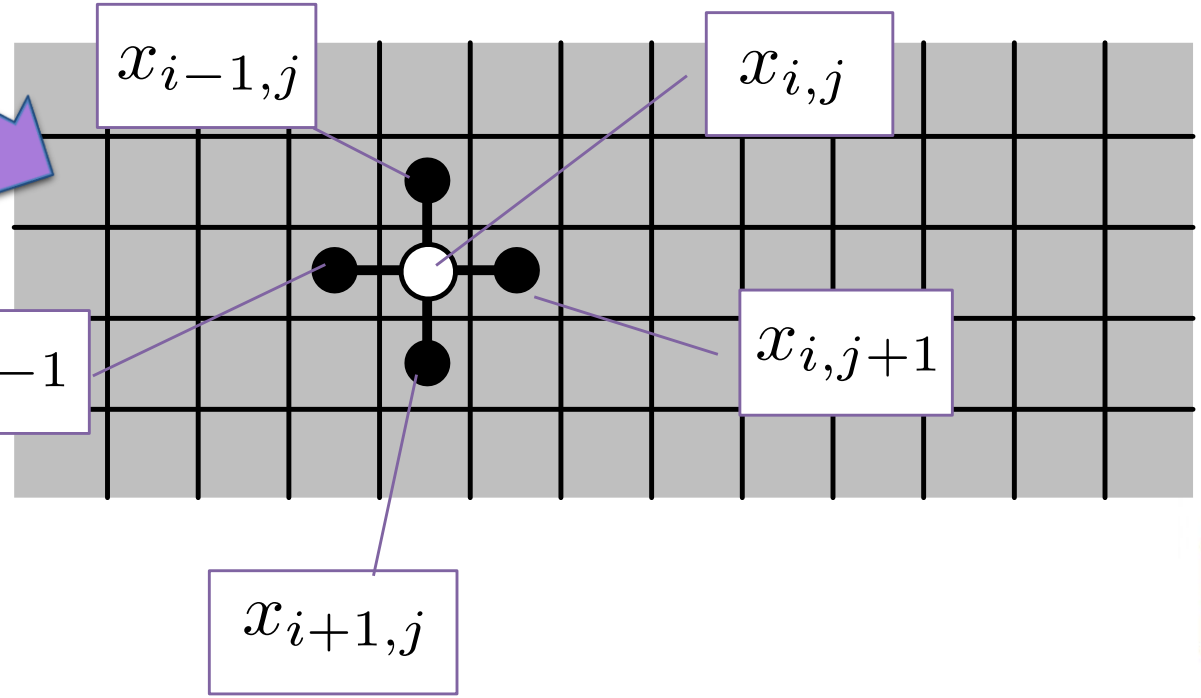
Laplace's Equation on a Regular Grid



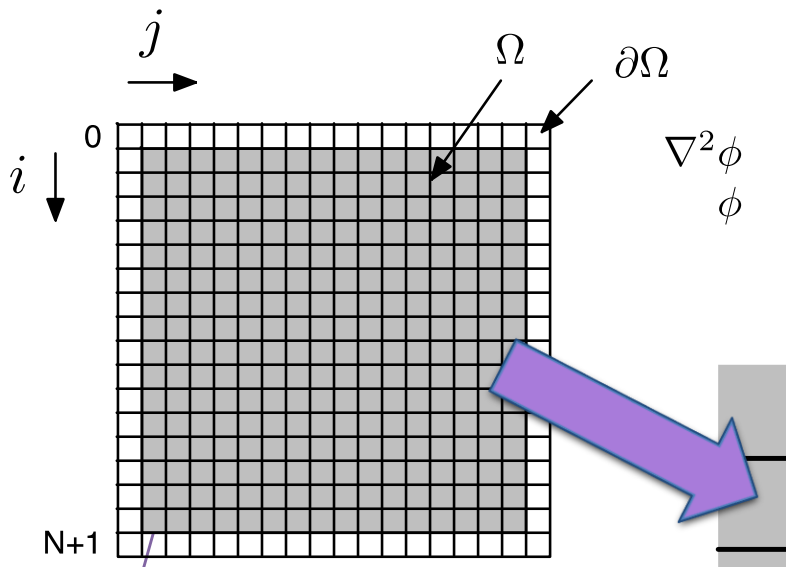
$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

k, k+1: Iteration timesteps

$$x_{i,j}^{k+1} = (x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k})/4$$



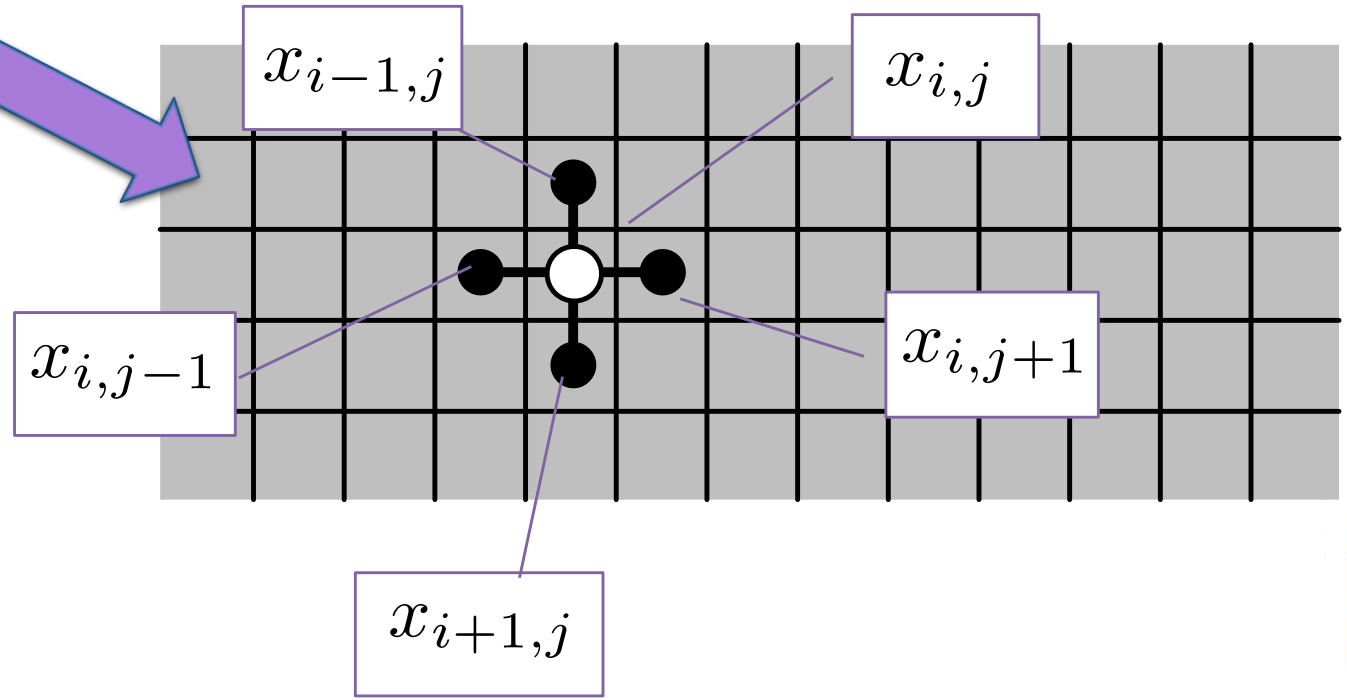
Laplace's Equation on a Regular Grid



$$\nabla^2\phi = 0$$

$$\phi = \text{given}$$

```
while (!converged) {
    for (size_t i = 1; i != N + 1; ++i) {
        for (size_t j = 1; j != N + 1; ++j) {
            y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                       x(i, j - 1) + x(i, j + 1)) / 4.0;
    swap(x, y);
}
```

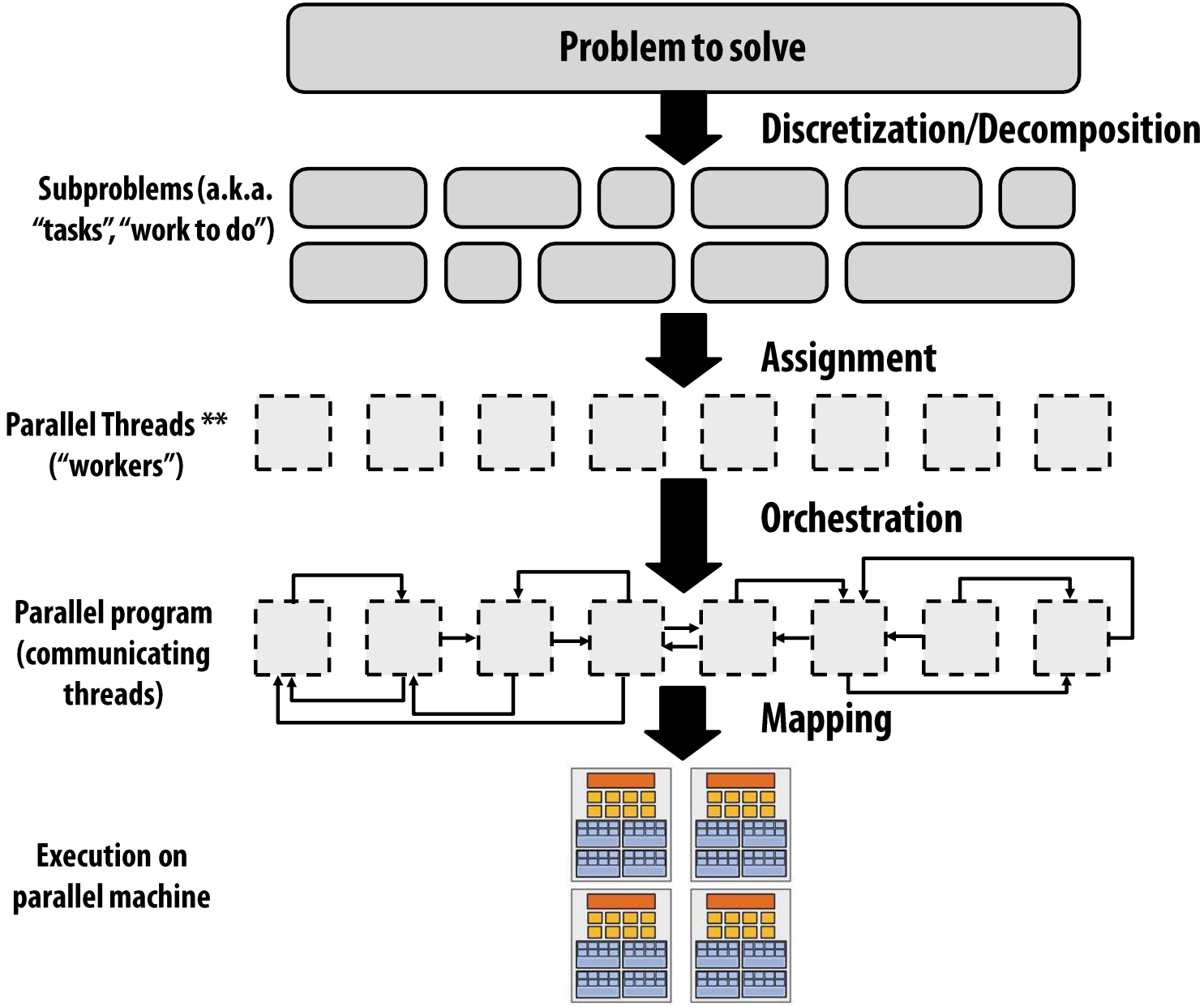


Main Sequential Jacobi Sweep

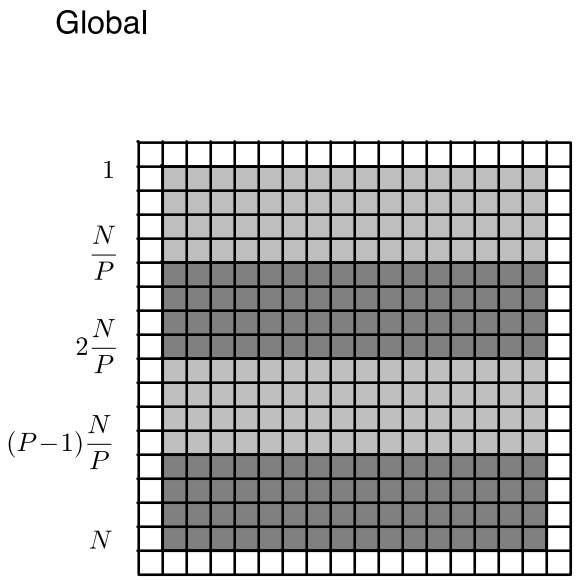
```
using grid = Matrix;

double jacobi_step(grid const& x, grid& y)
{
    assert(x.rows() == y.rows() && x.columns() == y.columns());
    double rnorm = 0.0;

    for (size_t i = 1; i != x.rows() - 1; ++i)
    {
        for (size_t j = 1; j != y.columns() - 1; ++j)
        {
            y(i, j) =
                (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) + x(i, j + 1)) / 4.0;
            rnorm += sqr(y(i, j) - x(i, j));
        }
    }
    return std::sqrt(rnorm);
}
```

Sequential Jacobi Solver

```
int jacobi(grid&x0, grid& x1, size_t max_iterations, double epsilon)
{
    for (size_t step = 0; step != max_iterations; ++step)
    {
        double rnorm = jacobi_step(x0, x1);
        if (rnorm < epsilon) return 0;
        std::swap(x0, x1);
    }
    return -1;
}
```

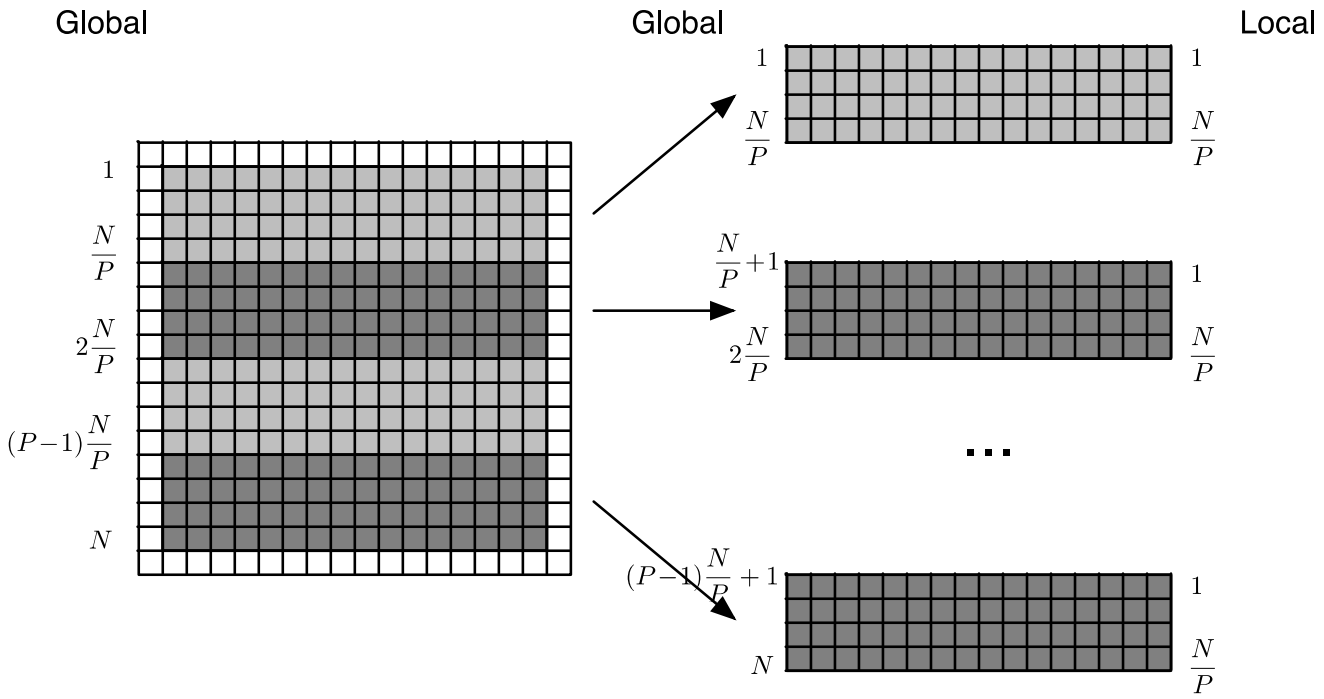

Problem to solve
Discretization/Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

Decomposition




```
for (size_t i = 1; i != N - 1; ++i)
    for (size_t j = 1; j != N - 1; ++j)
        y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                   x(i, j - 1) + x(i, j + 1)) / 4.0;
```

Decomposition

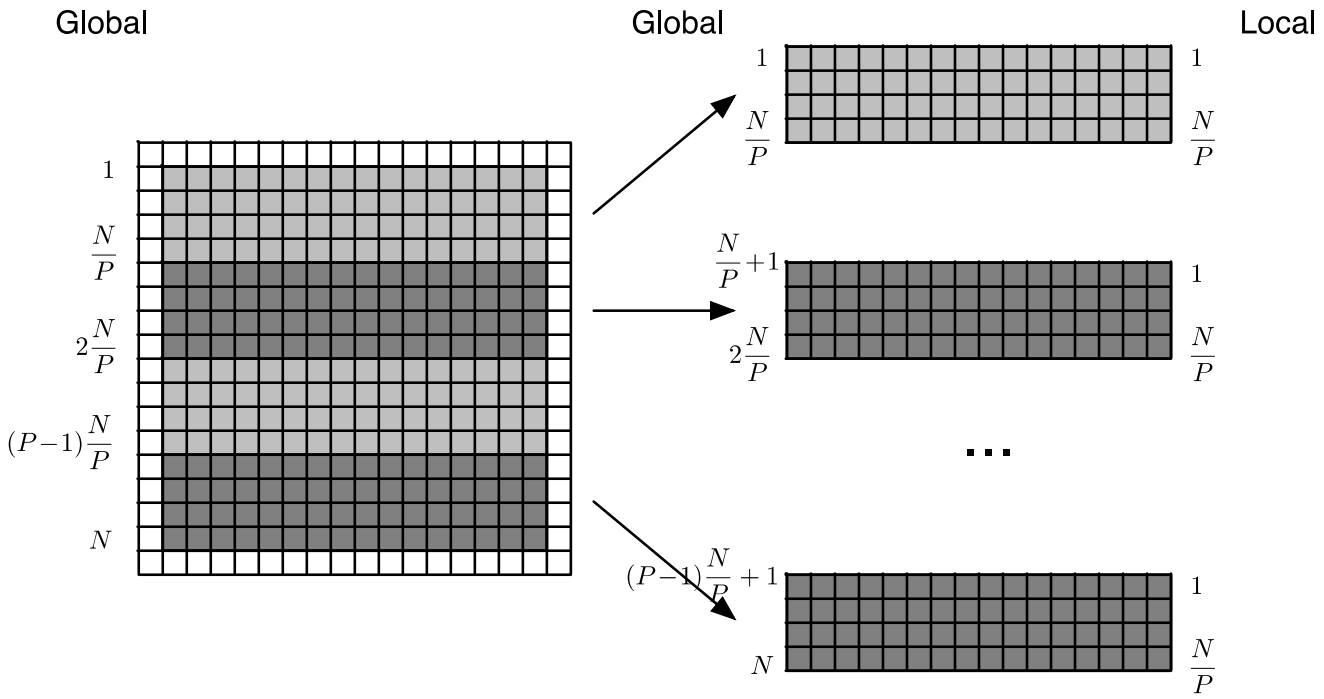



```
for (size_t i = 1; i != N - 1; ++i)
    for (size_t j = 1; j != N - 1; ++j)
        y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                   x(i, j - 1) + x(i, j + 1)) / 4.0;
```

Decomposition

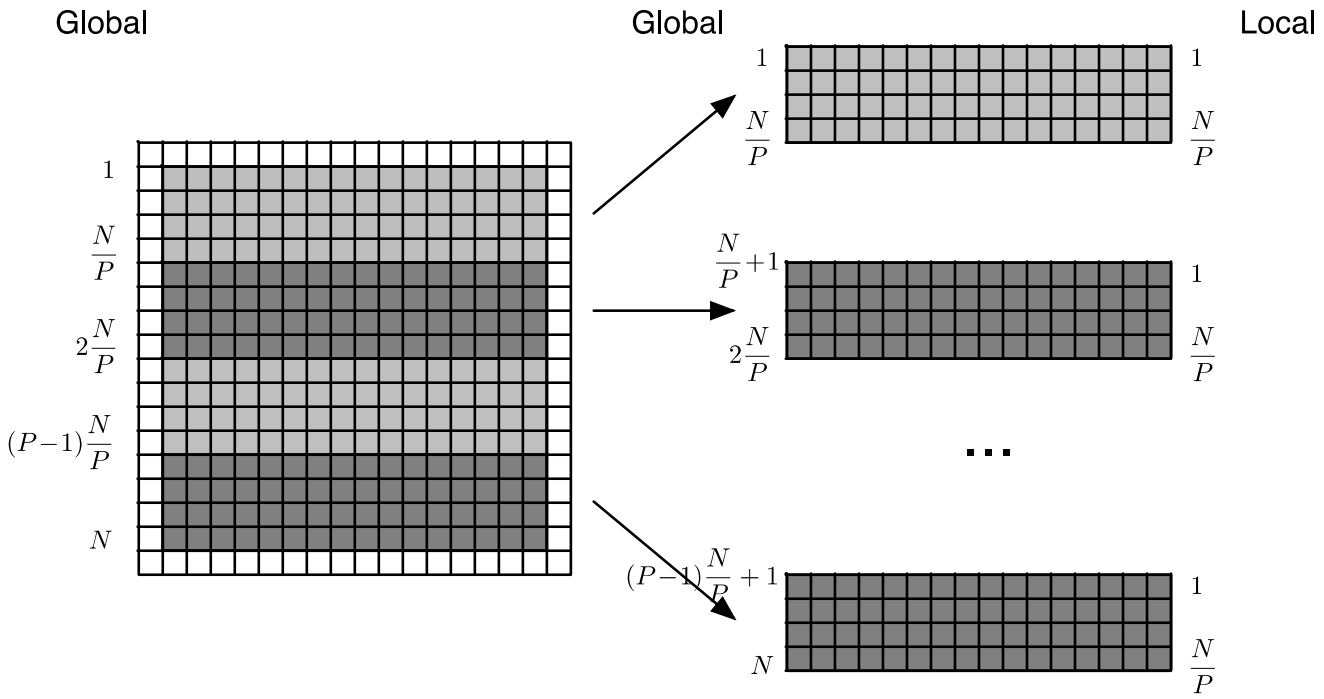
- Global partitioned index space (from 1 to N)
- SPMD index space
  - Identical for all partitions
  - Each partition $\boldsymbol{K}$ in original problem covers $K * N/P$ to $(K + 1) * N/P$ rows
- Decompose entire grid into $\boldsymbol{P}$ partitions
  - We aren't storing entire array on each node
  - We are storing only N/P rows
  - Inner loop still iterates 1 to N

Decomposition (global indices)




```
for (size_t i = K * N/P + 1; i != (K + 1) * N/P - 1; ++i)
    for (size_t j = 1; j != N - 1; ++j)
        y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                   x(i, j - 1) + x(i, j + 1)) / 4.0;
```

Decomposition (local indices)

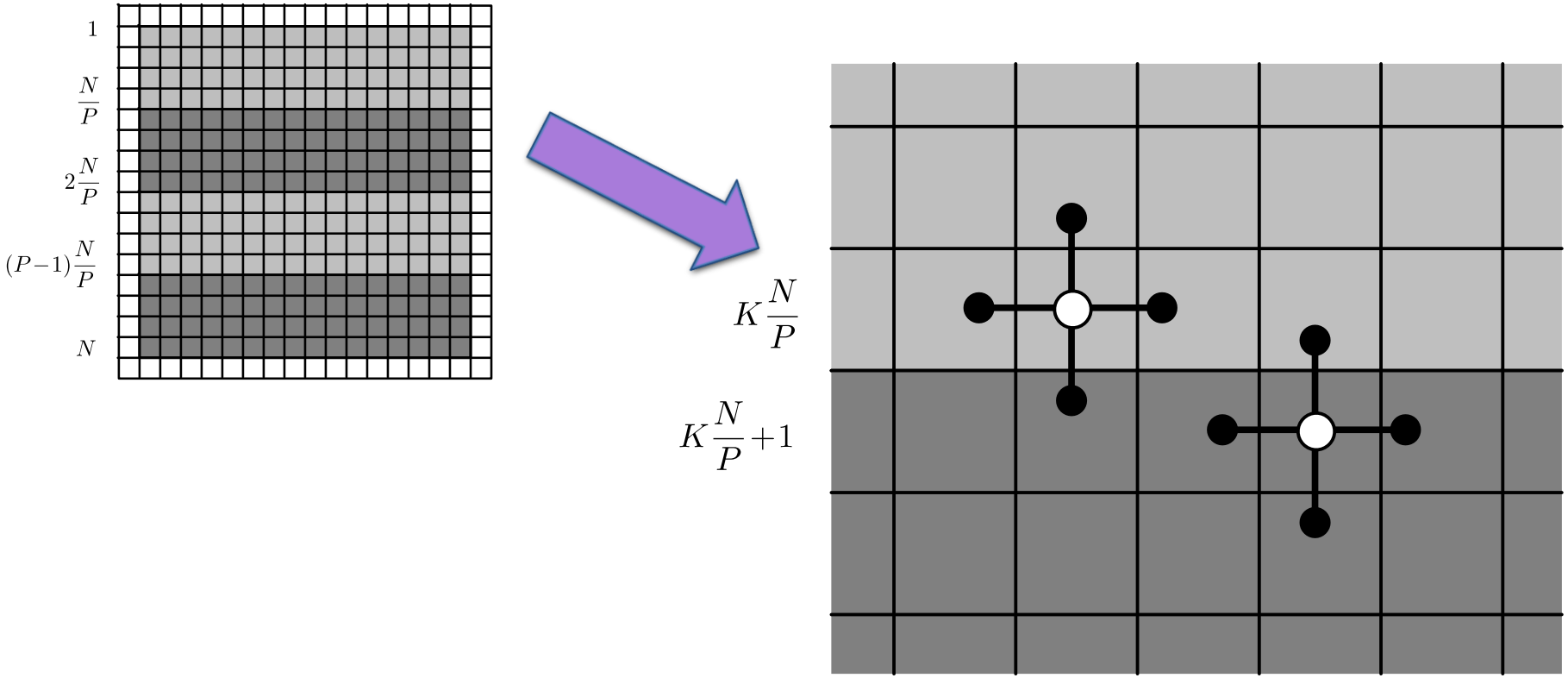



```
for (size_t i = 1; i != N/P - 1; ++i)
    for (size_t j = 1; j != N - 1; ++j)
        y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                   x(i, j - 1) + x(i, j + 1)) / 4.0;
```

Decomposition

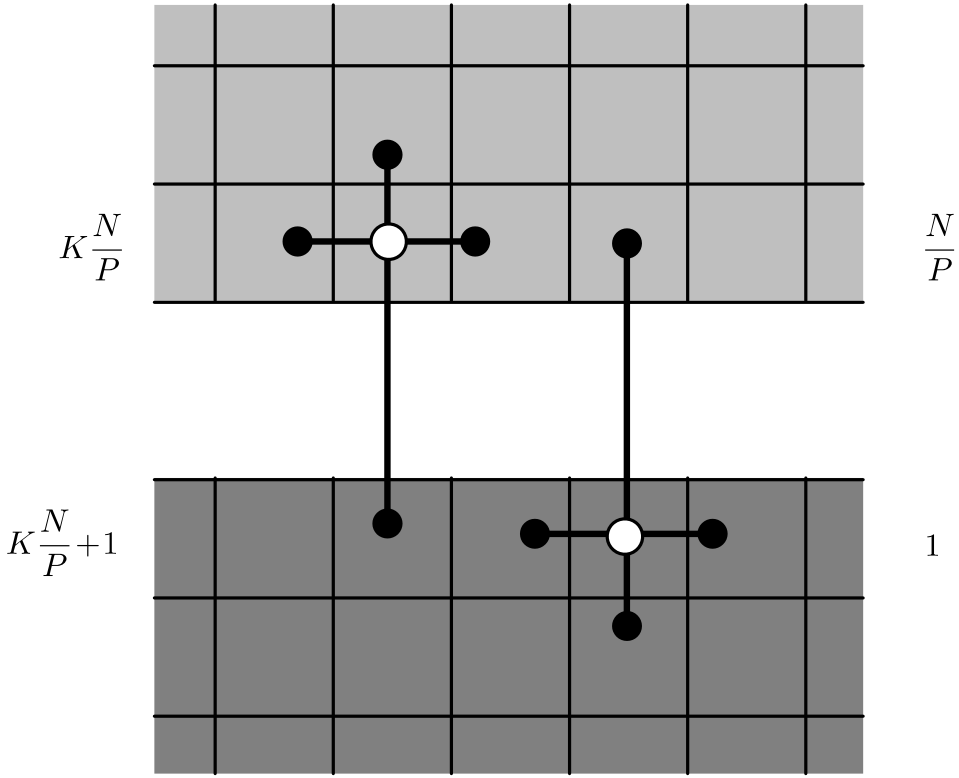
- Global partitioned index space (from 1 to N)
- SPMD index space
  - Identical for all partitions
  - Each partition $\boldsymbol{K}$ in original problem covers $K * N/P$ to $(K + 1) * N/P$ rows
- Decompose entire grid into $\boldsymbol{P}$ partitions
  - We aren't storing entire array on each node
  - We are storing only N/P rows
  - Inner loop still iterates 1 to N
- But what are the problems with that?
  - Data dependencies for stencil crosses partition boundary in original problem

Decomposition



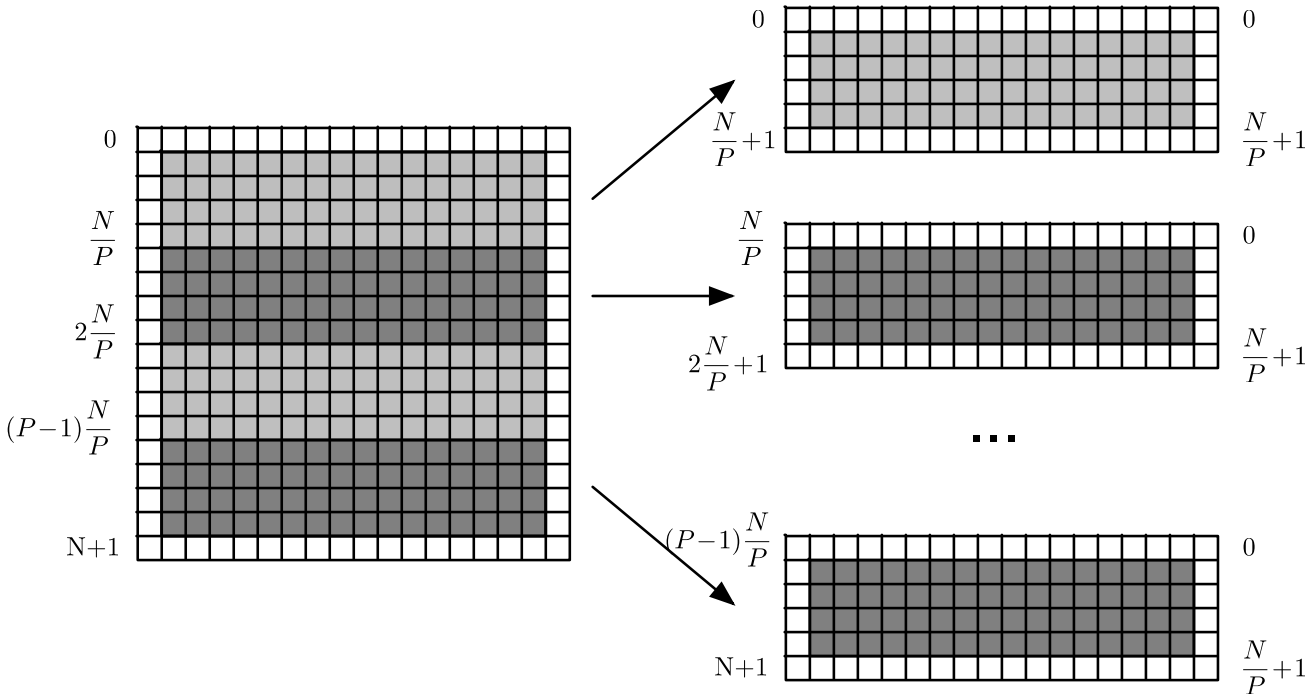
Decomposition

- The code reads/writes local data
  - Stencil accesses elements from neighboring partitions
- Problem in distributed memory
  - Causes read to non-existing (local) data
- We need to make these into valid reads
  - And preserve the as-if property
- Add an extra row (on some nodes) for the boundary
  - Needed for as-if

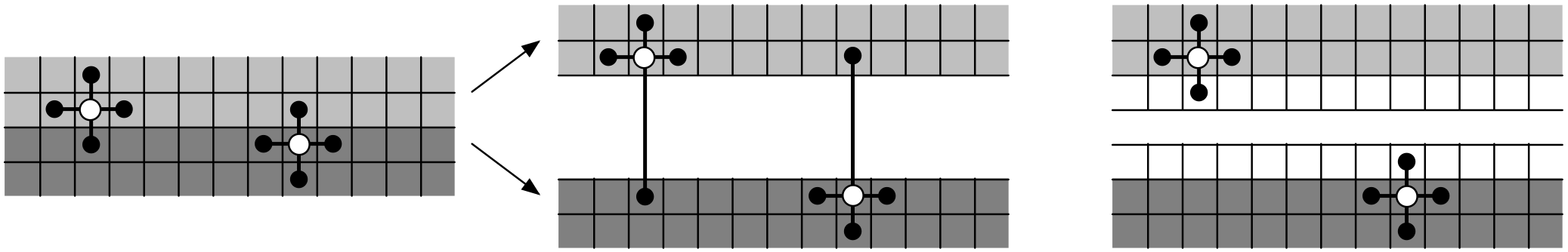


Decomposition

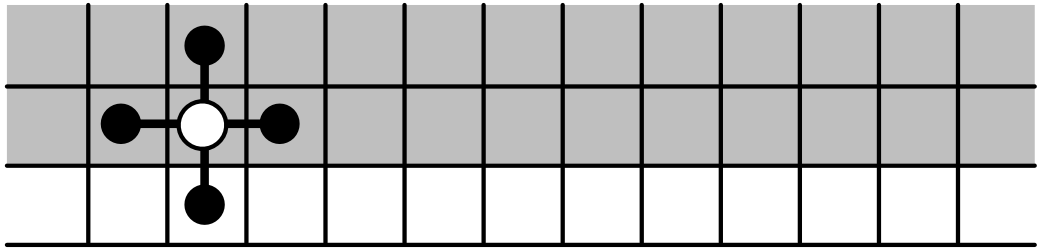
- Add a boundary to each of the partitions
  - Not part of the original problem
  - But needed to create the (as-if) illusion
  - To the local / SPMD code, the boundary and as-if are the same



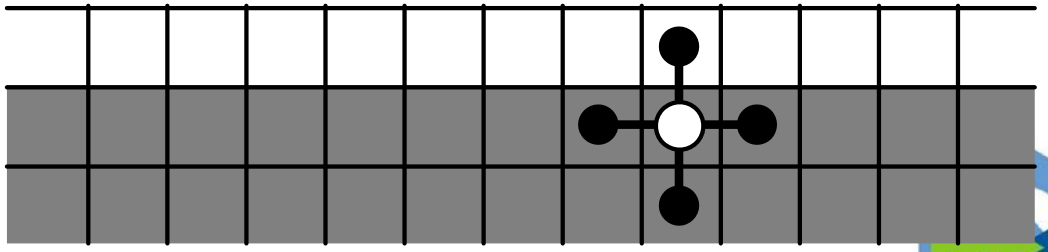
As-If



- Last row of upper partition needs to be as-if it was the second row of lower partition



- First row of lower partition needs to be as-if it was the second to the last row of lower partition



As-If

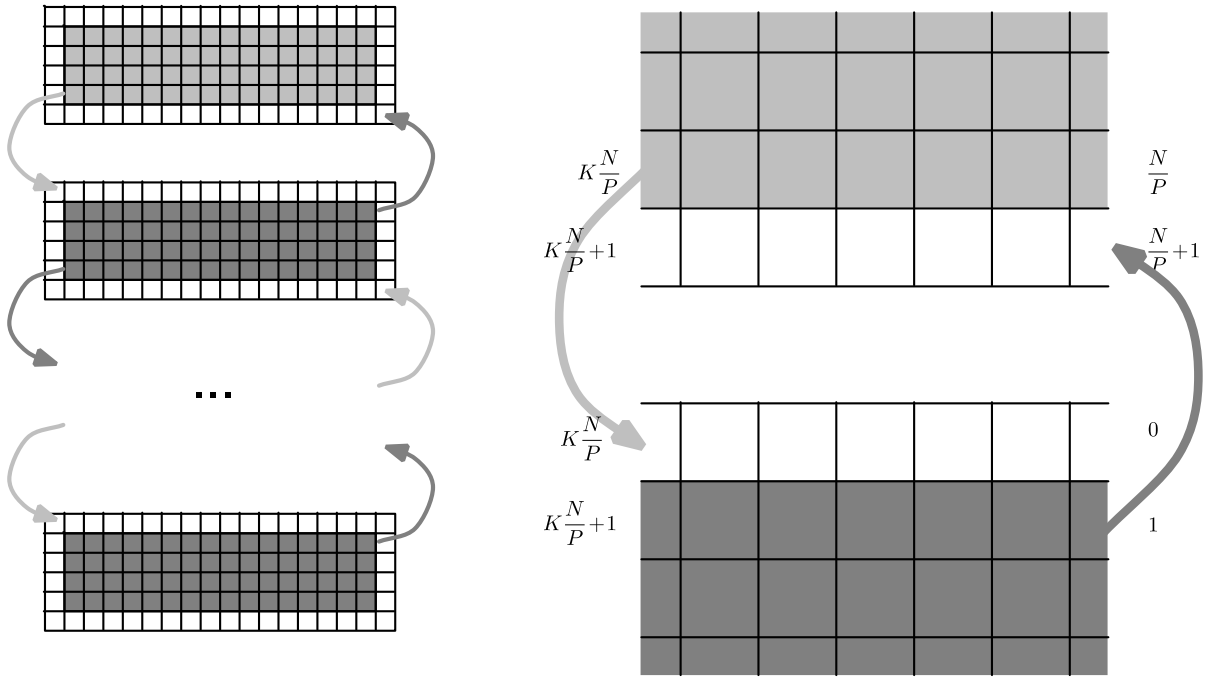
- Note that the additional boundary rows are not changed during an iteration
- We access elements of the boundary only for reading

```
for (size_t i = 1; i != N/P - 1; ++i)
    for (size_t j = 1; j != N - 1; ++j)
        y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                   x(i, j - 1) + x(i, j + 1)) / 4.0;
```

- The grid `y` is only written to, while the grid `x` is only read from
- Does the boundary row always have to have the same value as the other row?
  - No, only for the duration of one iteration
  - The boundary changes only on every outer iteration (on the swap())
  - We need to make as-if true after the swap

Compute / Communicate

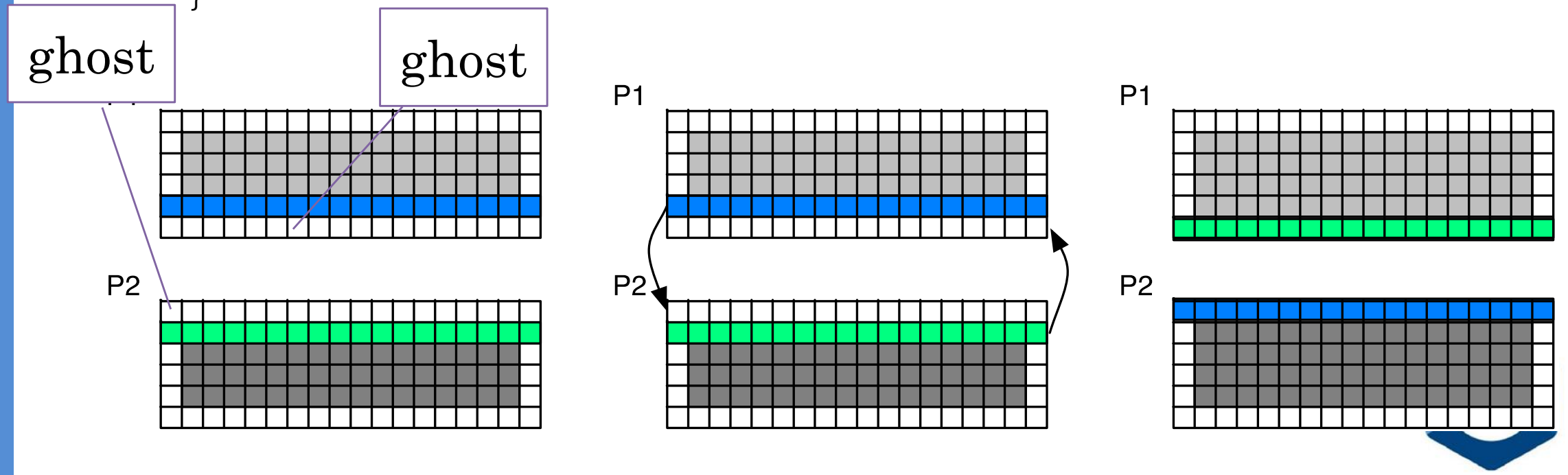
- To ensure “as-if”,
  - We need to update the boundary cells with “as-if” values from the neighboring partition
  - Before they are read at the next outer iteration



Compute / Communicate

```
while (!converged) {
    for (size_t i = 1; i != N + 1; ++i) {
        for (size_t j = 1; j != N + 1; ++j) {
            y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                       x(i, j - 1) + x(i, j + 1)) / 4.0;
    swap(x, y);
    make_as_if(x);  // collectively communicate ghost cells
}
```

Standard terminology for as-if boundary is ghost cell



Compute / Communicate

- The alternating sequence of compute and communicate is an almost universal pattern in SPMD programs

Compute

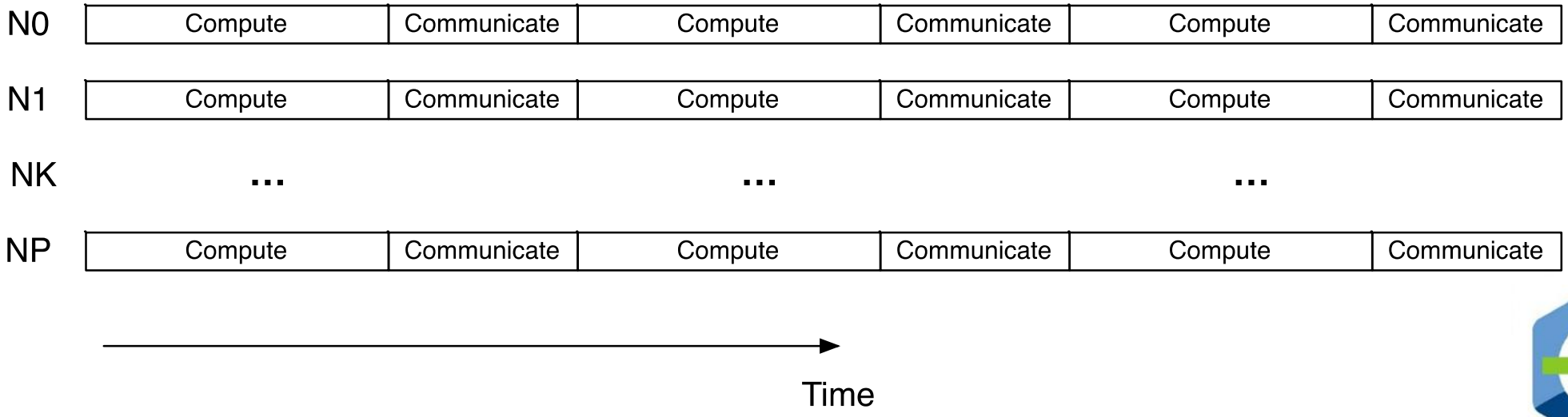
Communicate

```
while (!converged) {
    for (size_t i = 1; i != N + 1; ++i) {
        for (size_t j = 1; j != N + 1; ++j) {
            y(i, j) = (x(i - 1, j) + x(i + 1, j) +
                       x(i, j - 1) + x(i, j + 1)) / 4.0;
    swap(x, y);
    make_as_if(x);  // communicate ghost cells (collectively)
}
```

- As we will see, this is also one of the universal performance problems with SPMD program as well
  - Especially if required computation is not uniform across localities
  - Always waits for the 'slowest guy' before proceeding to next timestep

Compute / Communicate

- "Bulk Synchronous Parallel" (BSP) programming model
  - Processors are still only loosely coupled
  - But the compute / communicate pattern keeps them (collectively) synched in a bulk sense (after each time step)
- Performs and scales reasonably well as long as computation is load balanced
  - However: always uses only 'half' of the machine



Parallel Jacobi Solver

```
int jacobi(grid&x0, grid& x1, size_t max_iterations, double epsilon)
{
    for (size_t step = 0; step != max_iterations; ++step)
    {
        double rnorm = jacobi_step(x0, x1);
        if (rnorm < epsilon) return 0;
        std::swap(x0, x1);
    }
    return -1;
}
```

As-if: This needs to happen on all nodes, use `all_reduce` instead of `reduce`

As-if: Update ghost cells

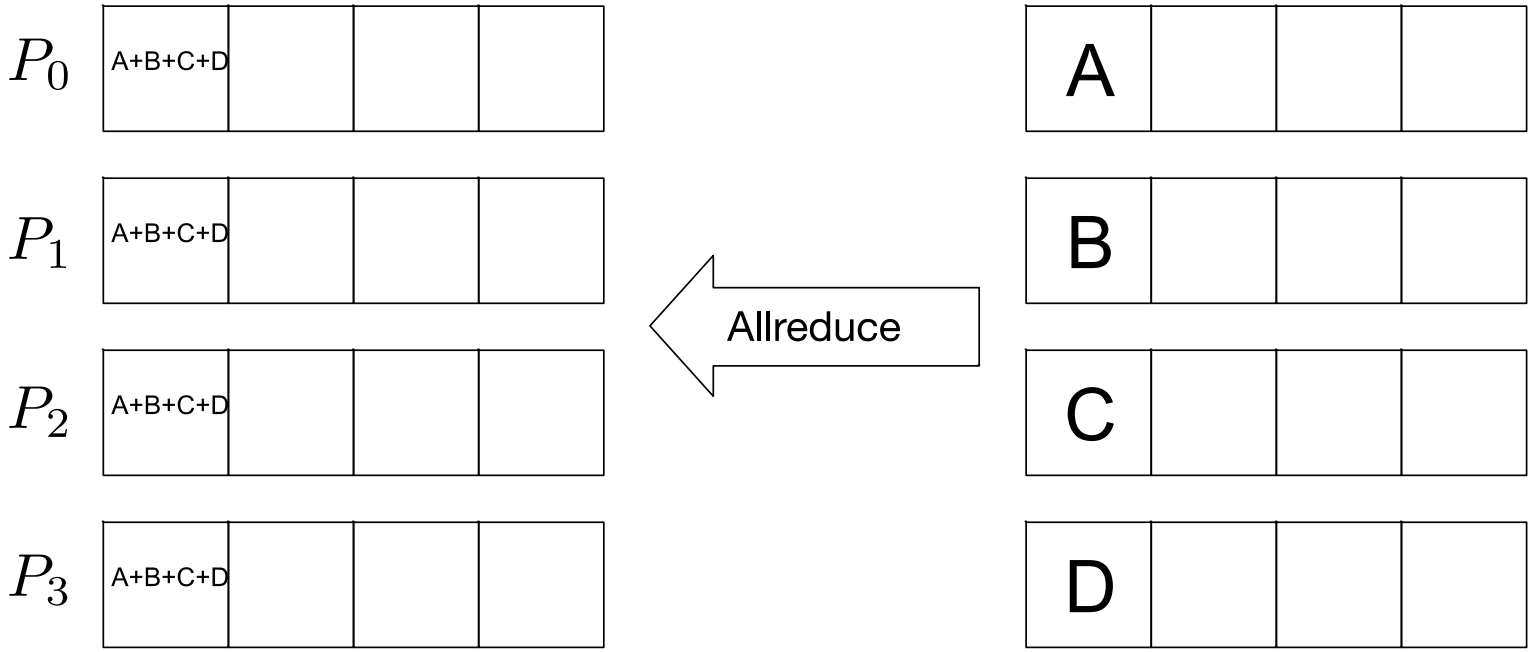
Main Parallel Jacobi Step

```
using grid = Matrix;

double jacobi_step(grid const& x, grid& y)
{
    assert(x.rows() == y.rows() && x.columns() == y.columns());
    double rnorm = 0.0;

    for (size_t i = 1; i != x.rows() - 1; ++i)
    {
        for (size_t j = 1; j != y.columns() - 1; ++j)
        {
            y(i, j) =
                (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) + x(i, j + 1)) / 4.0;
            rnorm += sqr(y(i, j) - x(i, j));
        }
    }
    return std::sqrt(rnorm);
}
```

No changes!

hpx::collectives::all_reduce

```
template <typename T, typename F>
T all_reduce(hpx::launch::sync_policy, communicator comm,
    T&& value, F&& op,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```



`hpx::collectives::all_reduce`

- The value is the 'send buffer' for all localities
- After `all_reduce` returns, all localities have the reduction result of the values supplied by all localities

- Note (true for all collective operations):
  - All localities connected to the given communicator must call the function
  - Otherwise none of the localities will finish the operation

hpx::collectives::all_reduce

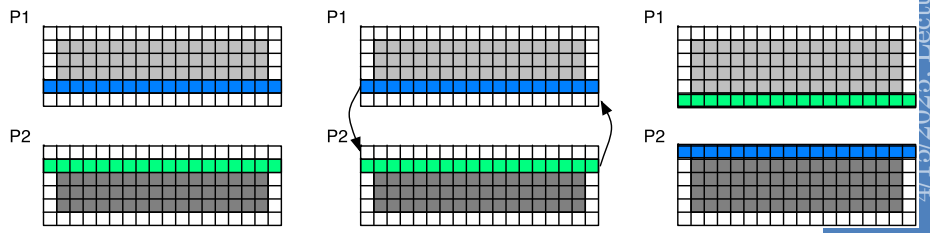
```
template <typename T, typename F>
T all_reduce(hpx::launch::sync_policy, communicator comm,
    T&& value, F&& op,
    this_site_arg this_site = this_site_arg(),
    generation_arg generation = generation_arg());
```

- Here:
  - `sync_policy`: special predefined type that instructs to execute the `all_reduce` operation synchronously (wait for operation to finish)
  - `comm`: the communicator instance used for this message
  - `value`: local data value to use for reduction
  - `this_site`: the local 'endpoint' (defaults to this locality)
  - `generation`: a sequence number of the operation (defaults to invocation counter)
- Returns the reduced value from all participating localities

Parallel Jacobi Solver

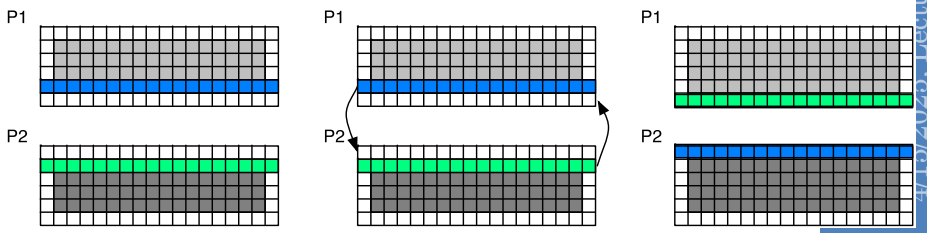
```
int jacobi_parallel(grid& x0, grid& x1, size_t max_iterations, double epsilon)
{
    for (size_t step = 0; step != max_iterations; ++step)
    {
        double local_norm = jacobi_step(x0, x1);

        double rnorm = all_reduce(hpx::launch::sync,
            hpx::collectives::get_world_communicator(), local_norm, std::plus{});

        if (rnorm < epsilon) return step;
        std::swap(x0, x1);

        update_ghosts(x0);    // maintain as-if, exchange ghost cells
    }
    return max_iterations;
}
```

Updating Ghost Cells

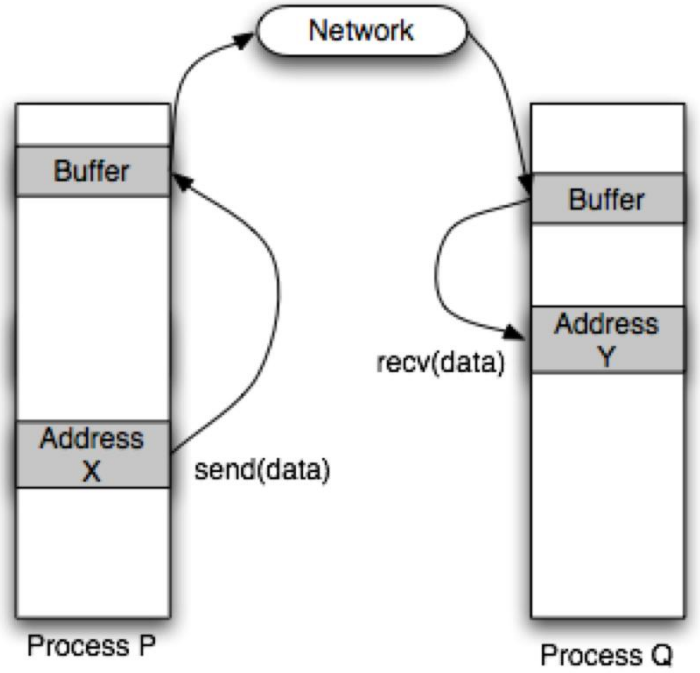


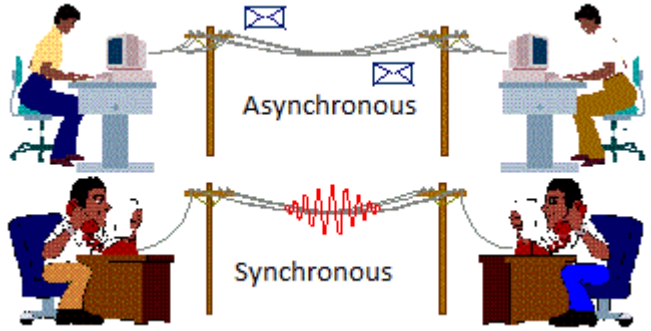

```
void update_ghosts(grid& x) {
    using ghost_cells = std::vector<double>;
    if (locality_id != 0) {
        // send our ghost cells to partition above
        set(hpx::launch::sync, channel_comm, that_site_arg(locality_id - 1),
            ghost_cells(row(x, 1)), tag_arg(2));

        // receive ghost cells from partition above
        row(x, 0) = get<ghost_cells>(hpx::launch::sync, channel_comm,
            that_site_arg(locality_id - 1), tag_arg(1));
    }

    if (locality_id != num_localities - 1) {
        // send our ghost cells to partition below
        set(hpx::launch::sync, channel_comm, that_site_arg(locality_id + 1),
            ghost_cells(row(x, x.rows() - 2)), tag_arg(1));

        // receive ghost cells from partition below
        row(x, x.rows() - 1) = get<ghost_cells>(hpx::launch::sync, channel_comm,
            that_site_arg(locality_id + 1), tag_arg(2));
    }
}
```

Updating Ghost Cells



- Why does that work?
- System will buffer the data
- The `set()` API will return once the channel has 'accepted the data
  - The corresponding `get<>()` does not need to have succeeded yet
- However, as we will see later, using asynchronous communication is beneficial (in many ways)

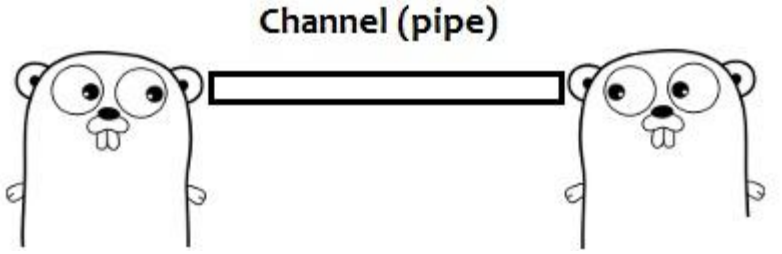




Asynchronous Communication

Asynchronous Channels

- High level abstraction of communication operations
  - Perfect for asynchronous boundary exchange
- Modelled after Go-channels
- Create on one thread, refer to it from another thread
  - Also: create on one locality (node), refer to it from another locality
  - Conceptually similar to bidirectional P2P communications
- Asynchronous in nature
  - `channel::get()` and `channel::set()` return futures



Asynchronous Jacobi (Take 1)

```
double jacobi_step_parallel_async(grid& x, grid& y) {
    auto comm = get_world_channel_communicator();
    auto [num_localities, locality_id] = comm.get_info();

    // exchange ghost cells with partition above and below
    std::vector<hpx::future<void>> ghost_updates;

    if (locality_id != 0) ghost_update_above(comm, x, ghost_updates);
    if (locality_id != num_localities - 1) ghost_update_below(comm, x, ghost_updates);

    double rnorm = jacobi_step(x0, x1); // local stencil application (without ghost cells)

    hpx::wait_all(ghost_updates);       // wait for boundary exchange to be finished

    return std::sqrt(rnorm + jacobi_step_ghosts());   // apply stencil to ghost cells
}
```

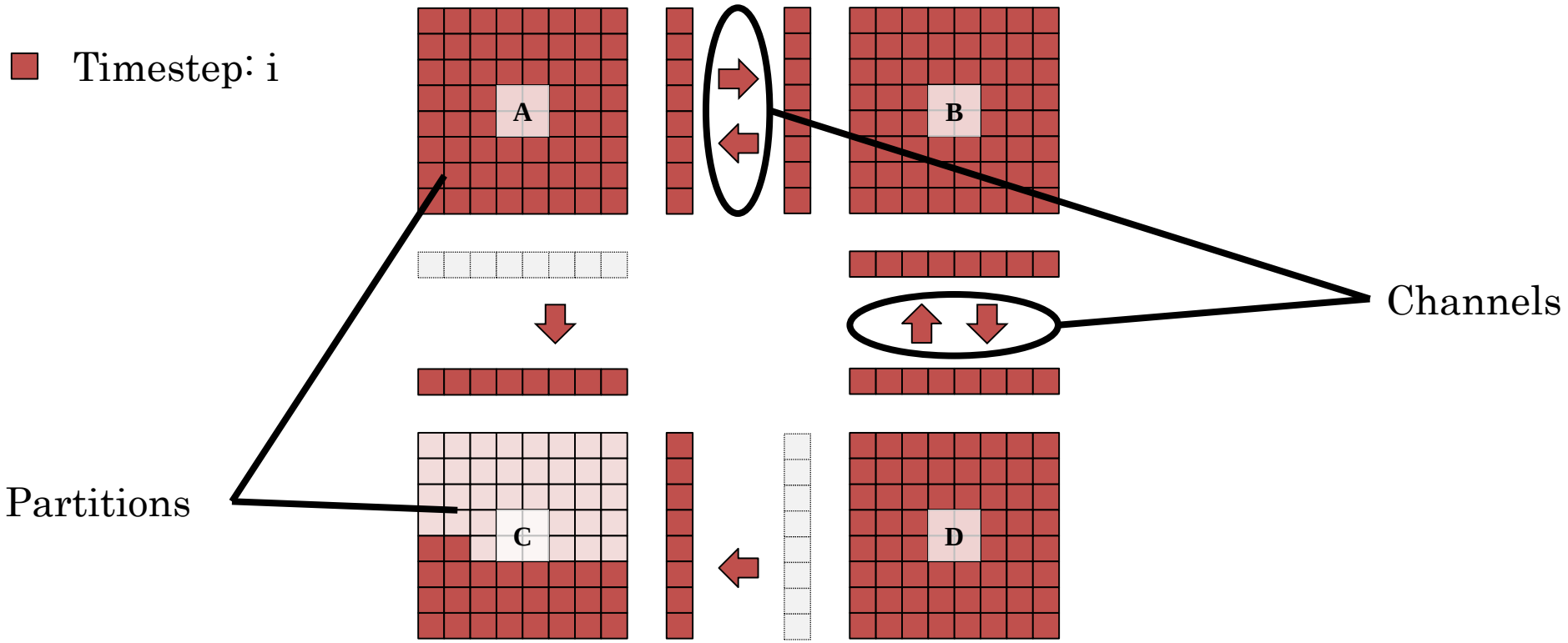
Asynchronous Jacobi (Take 1)

```
// exchange ghost cells with partition above
void ghost_update_above(hpx::collectives::channel_communicator comm, grid& x,
    std::vector<hpx::future<void>>& ghost_updates)
{
    using namespace hpx::collectives;

    auto [_, locality_id] = comm.get_info();

    // receive our ghost cells from partition above
    hpx::future<ghost_cells> recv =
        get<ghost_cells>(comm, that_site_arg(locality_id - 1), tag_arg(1));
    ghost_updates.emplace_back(
        recv.then([&](hpx::future<ghost_cells>&& f) { row(x, 0) = f.get(); }));

    // send our ghost cells to partition above
    ghost_updates.emplace_back(set(comm, that_site_arg(locality_id - 1),
        ghost_cells(row(x, 1)), tag_arg(2)));
}
```

Asynchronous Jacobi (Take 1)

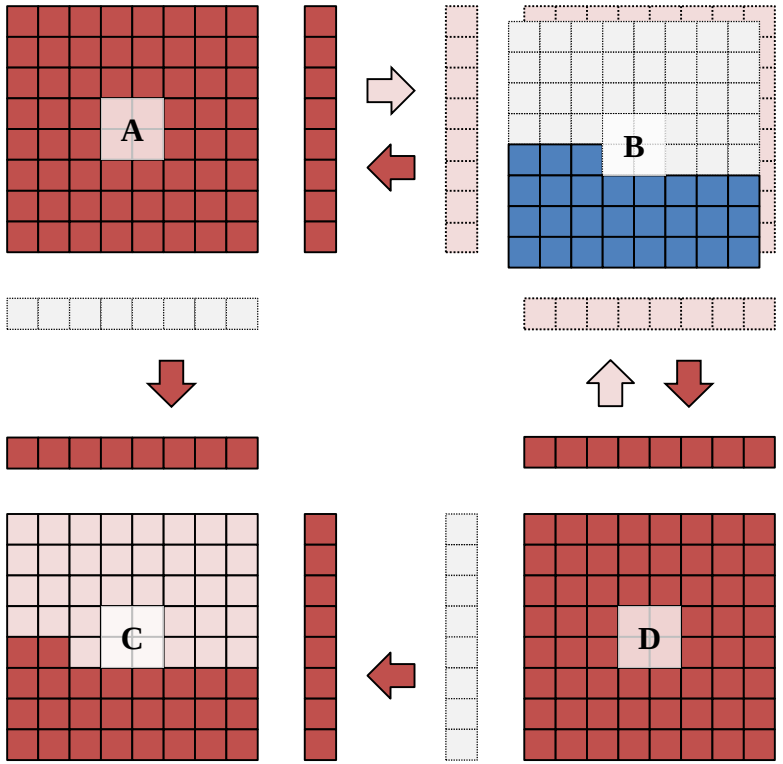
```
double jacobi_step_ghosts_row(grid const& x, grid& y, size_t i)
{
    double rnorm = 0.0;
    for (size_t j = 1; j != x.columns() - 1; ++j)
    {
        y(i, j) = (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) + x(i, j + 1)) / 4.0;
        rnorm += sqr(y(i, j) - x(i, j));
    }
    return rnorm;
}

// apply stencil to ghost cells (first and last rows)
double jacobi_step_ghosts(grid const& x, grid& y)
{
    return jacobi_step_ghosts_row(x, y, 1) +
        jacobi_step_ghosts_row(x, y, x.rows() - 2);
}
```

Futurized 2D Jacobi: Timestep i



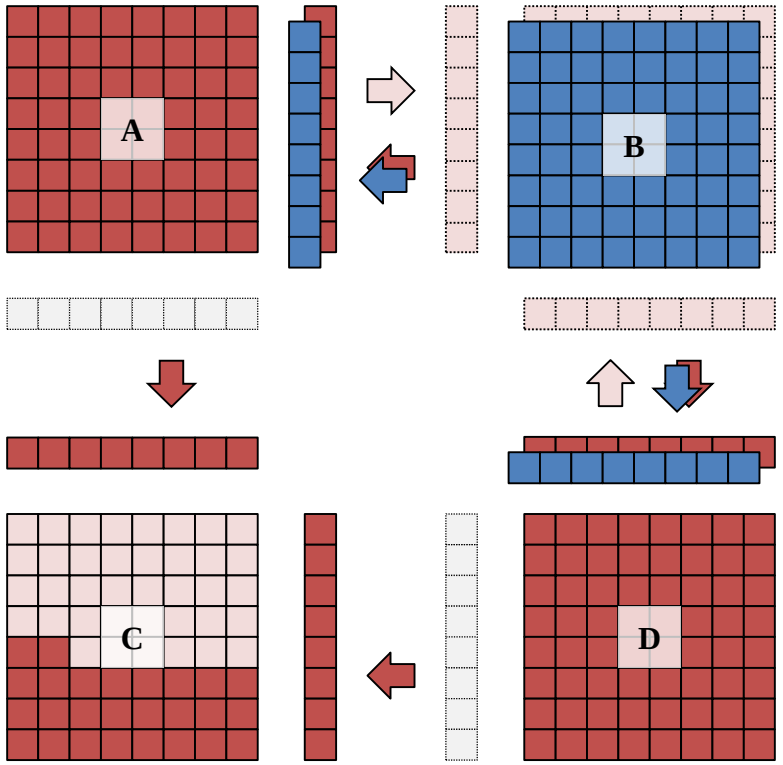
Futurized 2D Jacobi : Timestep i+1

- Timestep: i
- Timestep: i+1



Futurized 2D Jacobi

- Timestep: i
- Timestep: i+1



Futurized 2D Jacobi : Main Loop

```
// execute this for each partition concurrently
hpx::future<void> simulate(std::size_t steps)
{
    for (size_t t = 0; t != steps; ++t)
    {
        co_await perform_one_time_step(t);
    }
}
```

One Timestep: Update Boundaries

```
future<void> upper_boundary(int t);  // same for other boundaries

future<void> perform_one_time_step(int t)
{
    // Update our boundaries from neighbors
    co_await when_all(upper_boundary(t), right_boundary(t),
        lower_boundary(t), left_boundary(t));

    // Apply stencil to partition
    co_await for_loop(par(task), min + 1, max - 1,
        [&](size_t idx) { /* apply stencil to each inner point */ });
}
```

One Timestep: Ghost Cells

```
future<void> upper_boundary(int t)
{
    // Update upper boundary from upper neighbor
    vector<double> data = co_await channel_up_from.get(t);

    // process upper ghost-zone data using received data
    for_loop(seq, 1, size(data) - 1,
        [&](size_t idx) { /* apply stencil to each point in data */ });

    // send new ghost zone data back to upper neighbor
    co_await channel_up_to.set(data, t + 1);
}
```

2D Jacobi, Fully Parallelized

- Partitions are distributed across machine
- More partitions per node (locality) than cores
  - Oversubscription
- Code equivalent regardless whether neighboring partition is on the same node
- Overlap of communication and computation
  - More parallelism (work) than compute resources (cores)

Summary

- As-if is the most important principle in distributed parallelization (correctness first)
- SPMD has high degree of self-similarity – solving global problem is same as solving local problem – communication enforces as-if
- Ubiquitous compute / communicate cycle
- Adding asynchrony helps decoupling localities
- Adding local parallelism is orthogonal and straight-forward

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP