Synchronous & Asynchronous Computations

Thoai Nam

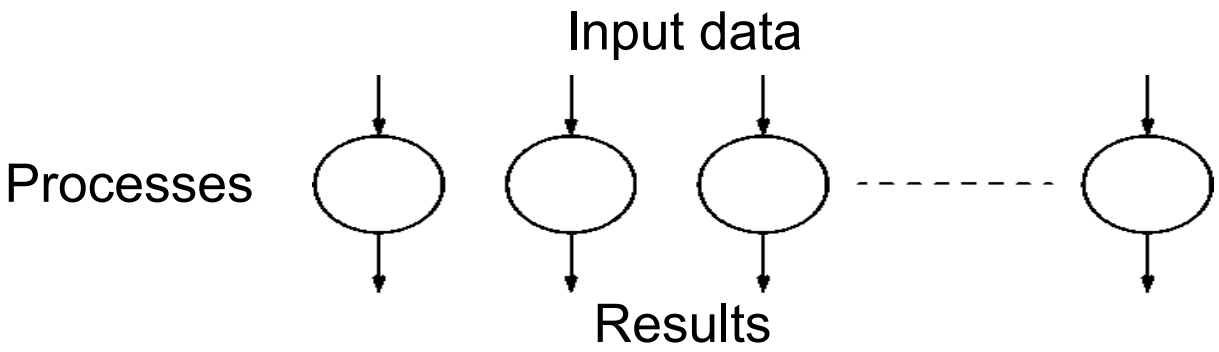
High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Technology

HCMC University of Technology

EPC

- A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or)



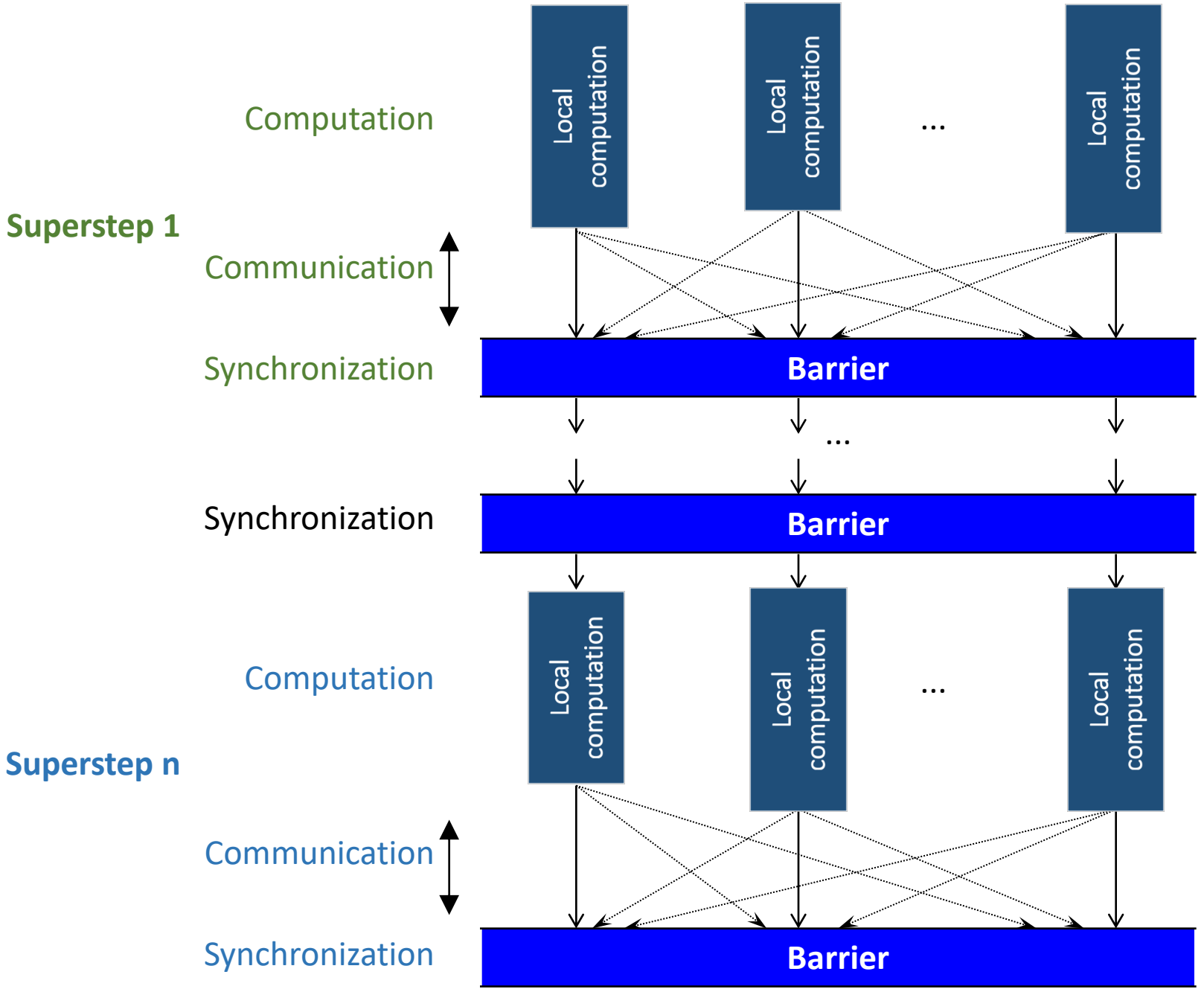
Communication???

Bulk Synchronous Parallel (BSP)

No communication or very little communication between processes;
Each process can do its tasks without any interaction with other processes.

- Data parallel computation

BSP


Computation
Local computation
Local computation
...
Local computation
Superstep 1
Communication
Synchronization
Barrier
...
Synchronization
Barrier
Computation
Local computation
Local computation
...
Local computation
Superstep n
Communication
Synchronization
Barrier

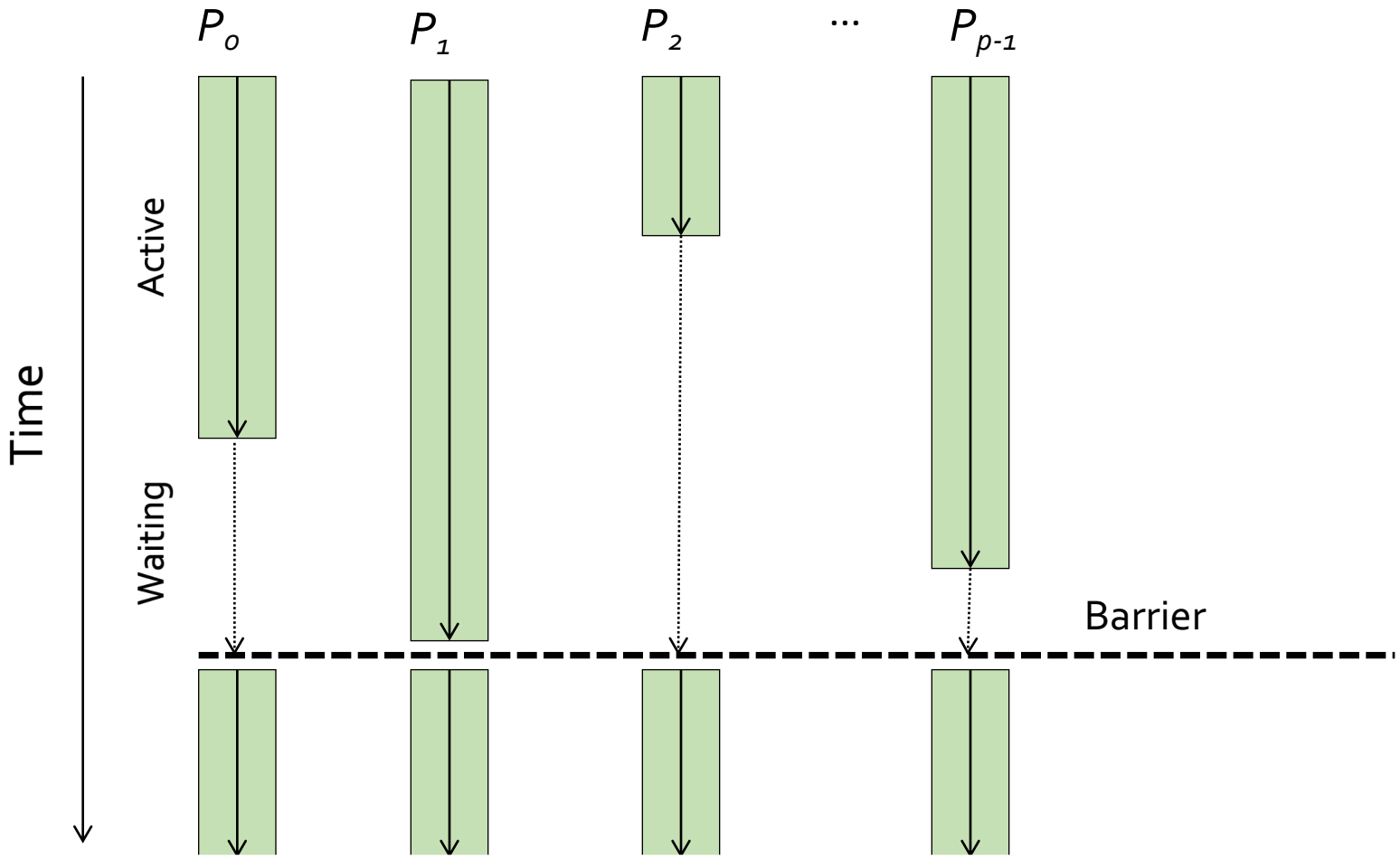
Synchronous computations

In a (fully) synchronous application, all the processes synchronized at regular points.

Barrier

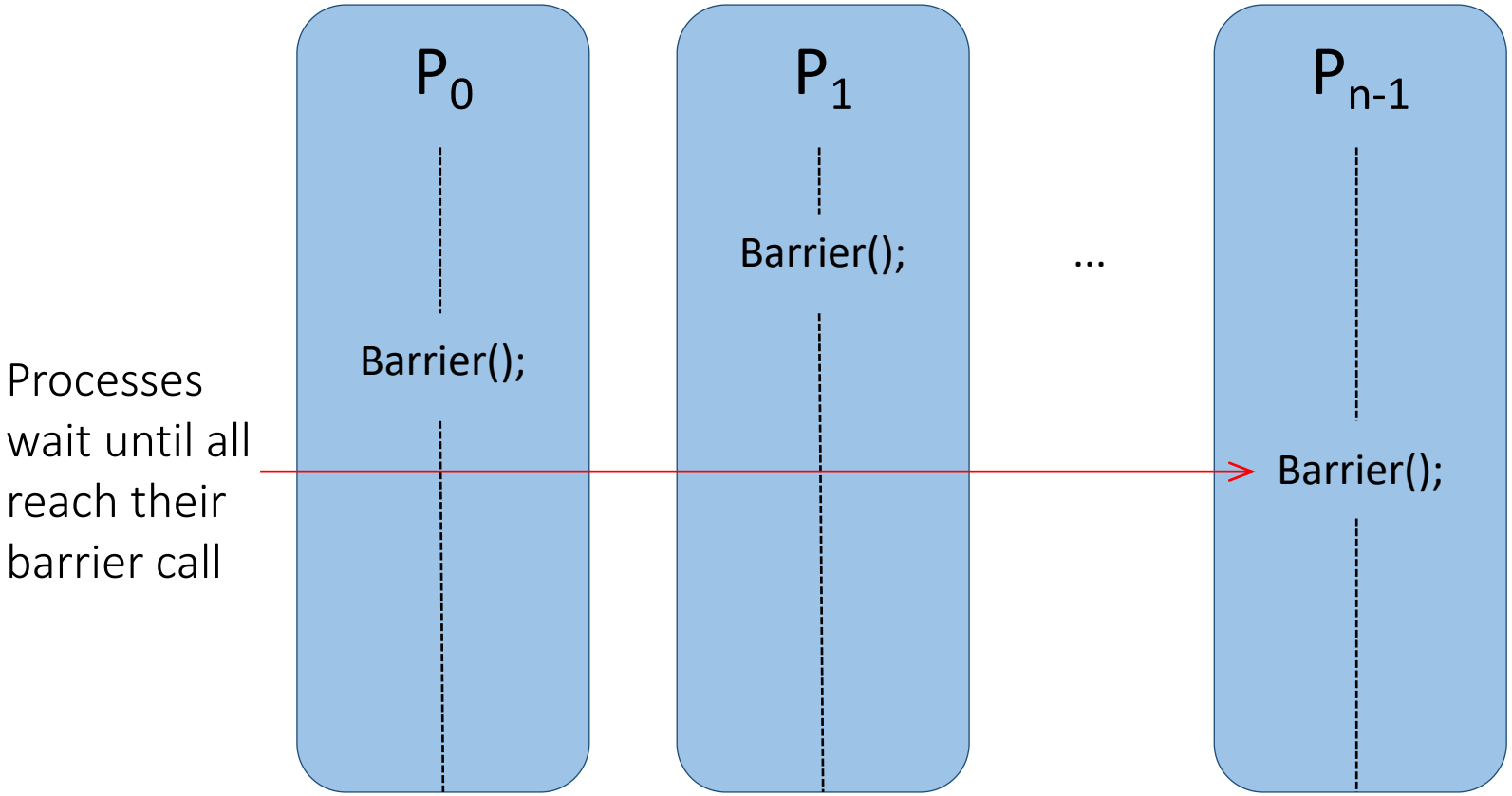
- A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait;
- All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

Processes reaching barrier at different times



Message-passing

- In message-passing systems, barriers provided with library routines



MPI

- MPI_Barrier()
- Barrier with a named communicator being the only parameter
- Called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then.

Synchronized Computations

Can be classified as:

- Fully synchronous

  In fully synchronous, all processes involved in the computation must be synchronized

- Locally synchronous

  In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

Fully synchronized computation examples

Data Parallel Computations

- Same operation performed on different data elements simultaneously; i.e., in parallel.

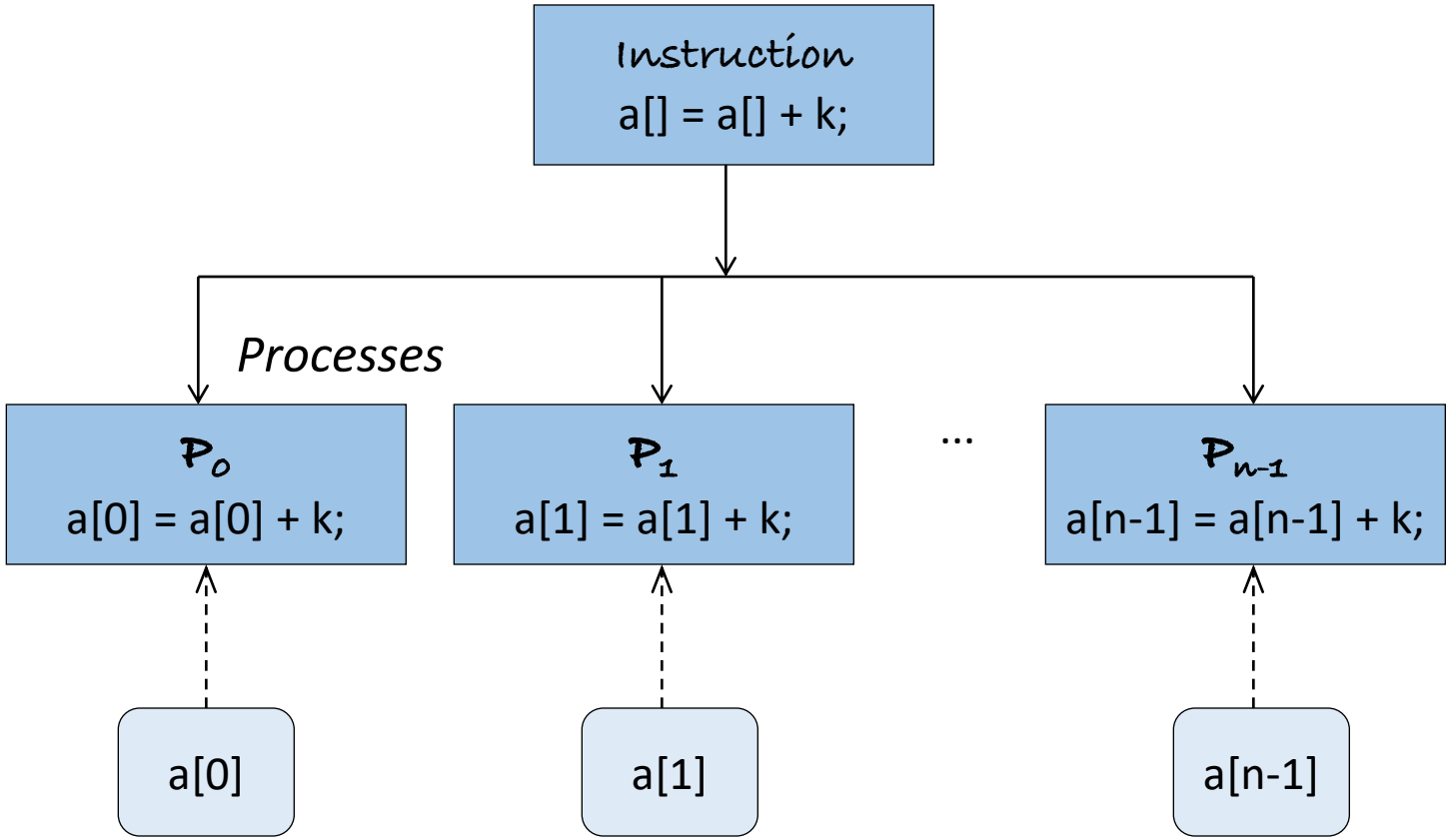
- Particularly convenient because:
  - Ease of programming (essentially only one program)
  - Can scale easily to larger problem sizes
  - Many numeric and some non-numeric problems can be cast in a data parallel form.

Example

- To add the same constant to each element of an array:

```
for (i=0; i<n; i++)
    a[i] = a[i] + k;
```

- The statement $a[i] = a[i] + k$ could be executed simultaneously by multiple processors, each using a different index i $(0 < i \leq n)$.



Forall construct

- Special “parallel” construct in parallel programming languages to specify data parallel operations:

```
forall (i=0; i<n; i++)
   S;
```

states that n instances of the statements of the body (S) can be executed simultaneously.

- One value of the loop variable i is valid in each instance of the body, the first instance has $i = 0$, the next $i = 1$, and so on.

Example

- To add **k** to each element of an array, **a**, we can write:

```
forall (i=0; i<n; i++)
    a[i] = a[i] + k;
```

Data parallel technique applied to multiprocessors and multicomputers.

- SPMD: to add **k** to the elements of an array:

```
i = Get_Rank();    // Pi có Rank=i với 0 ≤ i ≤ n-1
a[i] = a[i] + k;   // Thực thi S trong vòng lặp thứ i
Barrier(group_p); // Đồng bộ rào cản cho tất cả n tiến trình
```

Prefix sum problem

0:	x_0
1:	$x_0 \oplus x_1$
2:	$x_0 \oplus x_1 \oplus x_2$
	...
n-1:	$x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1}$

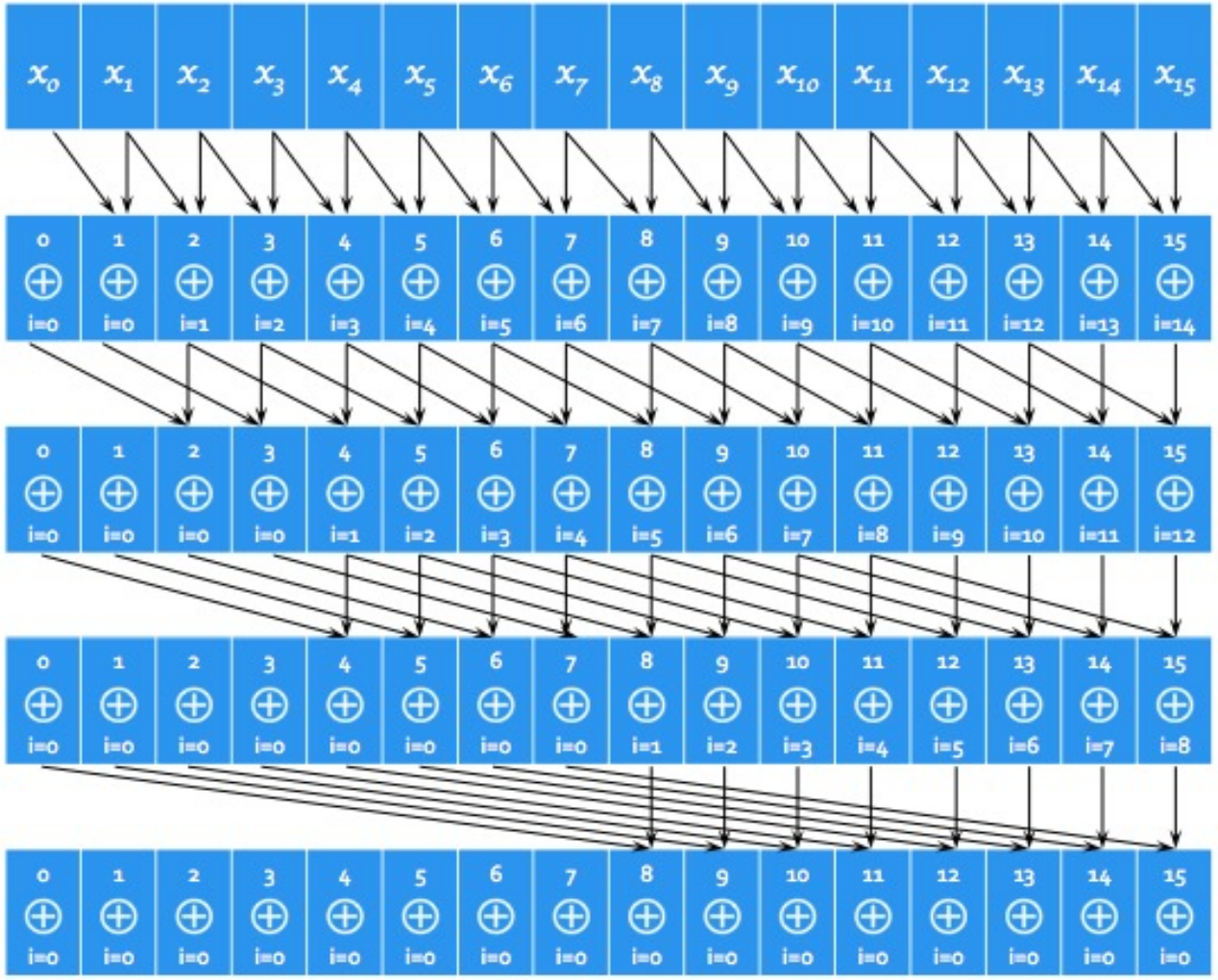
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	3	4	5	6	7	8

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	6	10	15	21	28	36

```
1.  Sequential_Prefix_sums (n, x[]) {
2.      for (int i = 1; i < n; i++)
3.          x[i] = x[i-1] ⊕ x[i];
4.      return x[];
5.  }
```

O(n)

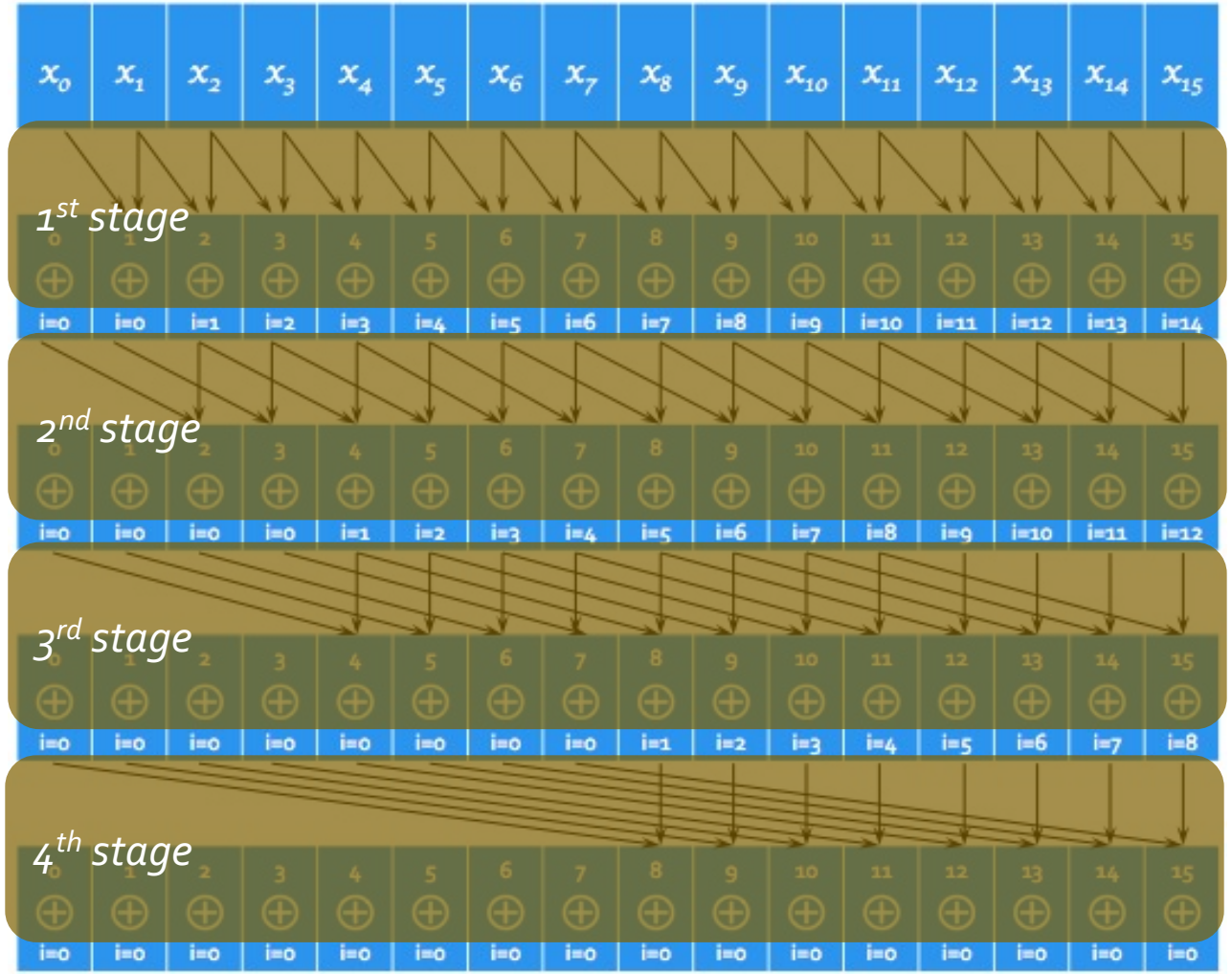
Data parallel example - prefix sum problem




```
1. Parallel_Prefix_sums (n, x[]) {
2.       for (int i = 0; i < ⌈log n⌉; i++)
3.           forall (int j = 0; j < n; j++)
4.               if (j ≥ 2^i)
5.                   x[j] = x[j] + x[j-2^i];
6.       return x[];
7.  }
```

O(logn)

Data parallel example - prefix sum problem




```
1. Parallel_Prefix_sums (n, x[]) {
2.       for (int i = 0; i < ⌈log n⌉; i++)
3.           forall (int j = 0; j < n; j++)
4.               if (j ≥ 2^i)
5.                   x[j] = x[j] + x[j-2^i];
6.       return x[];
7.   }
```

O(logn)

Synchronous Iteration (Synchronous Parallelism)

- Each iteration composed of several processes that start together at beginning of iteration. Next iteration cannot begin until all processes have finished previous iteration. Using **forall**:

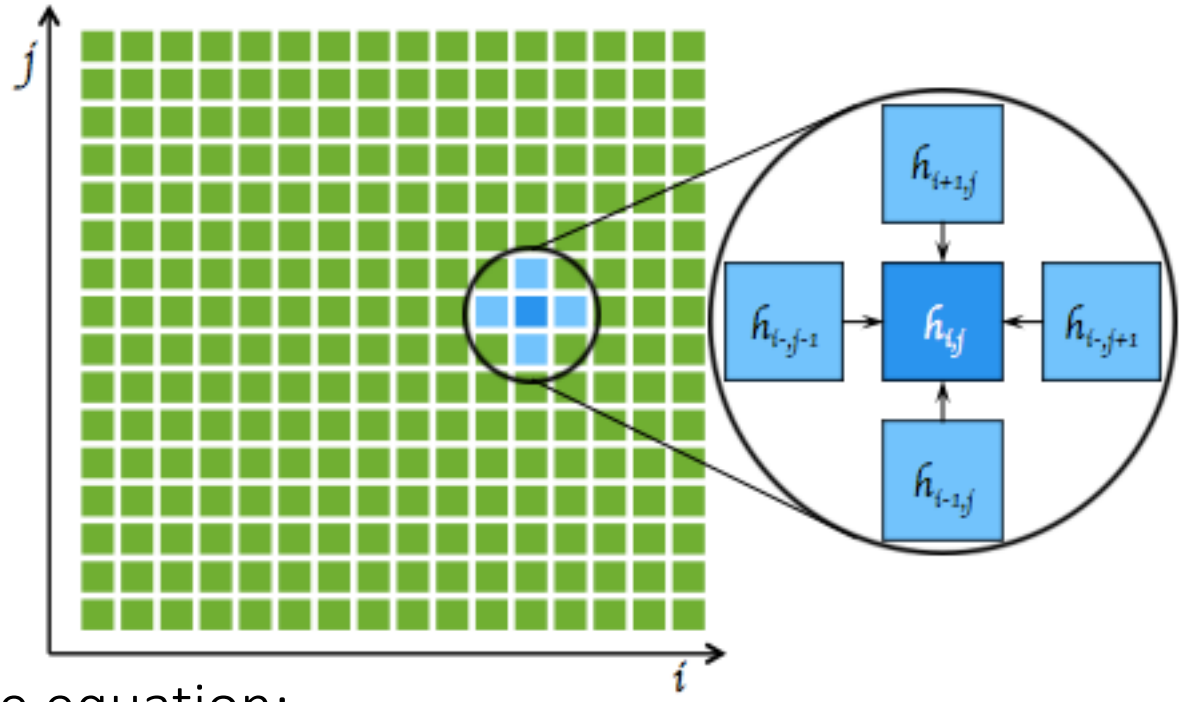
```
for (j=0; j<n; j++)        // Lặp n bước
  forall (i=0; i<p; i++) // p tiến trình thực hiện
      S(i);                // Công việc của mỗi tiến trình P_i
```

- SIMD:

```
for (j=0; j<n; j++) {  // Lặp n bước
  i = Get_Rank();    // P_i có rank=i với 0 ≤ i ≤ n-1
  S(i);              // Công việc của mỗi tiến trình P_i
  Barrier(group_p); // Đồng bộ rào cản cho tất cả p tiến trình
}
```

Heat distribution problem
(Locally synchronous computation)

- An area has known temperatures along each of its edges
- Find the temperature distribution within
- Divide area into fine mesh of points $h_{i,j}$. Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.



- Temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

($0 < i < n$, $0 < j < n$) for a fixed number of iterations or until the difference between iterations less than some very small amount.

Sequential algorithms

```
1. Seq_heat_distribution_ver1 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Cập nhật giá trị nhiệt mới tại bước k và h[i][j]
7.       for (i=1; i<n; i++)
8.         for (j=1; j<n; j++)
9.           h[i][j] = g[i][j];
   // Kiểm tra điều kiện kết thúc
10.      continue = false;
11.      for (i=1; i<n; i++)
12.        for (j=1; j<n; j++)
13.          if !converged(i, j) {
14.            continue = true;
15.            break;
16.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
17.   } while ((continue == true) && (k < (Max_loop-1)))
18. }
```

```
1. Seq_heat_distribution_ver2 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

Parallel algorithm

```
  // Lặp đến Max_loop
1.  for (k=0; k<Max_loop; k++) {
2.     h = 0.25 * (l + r + d + u);
  // Send() ở chế độ
  // không bị chặn (non-blocking)
3.     Send(&h, P_{i-1,j});
4.     Send(&h, P_{i+1,j});
5.     Send(&h, P_{i,j-1});
6.     Send(&h, P_{i,j+1});
  // Recv() ở chế độ hay đồng bộ
  // (synchronous) bị chặn (blocking)
7.     Recv(&l, P_{i-1,j});
8.     Recv(&r, P_{i+1,j});
9.     Recv(&d, P_{i,j-1});
10.    Recv(&u, P_{i,j+1});
11. }
```

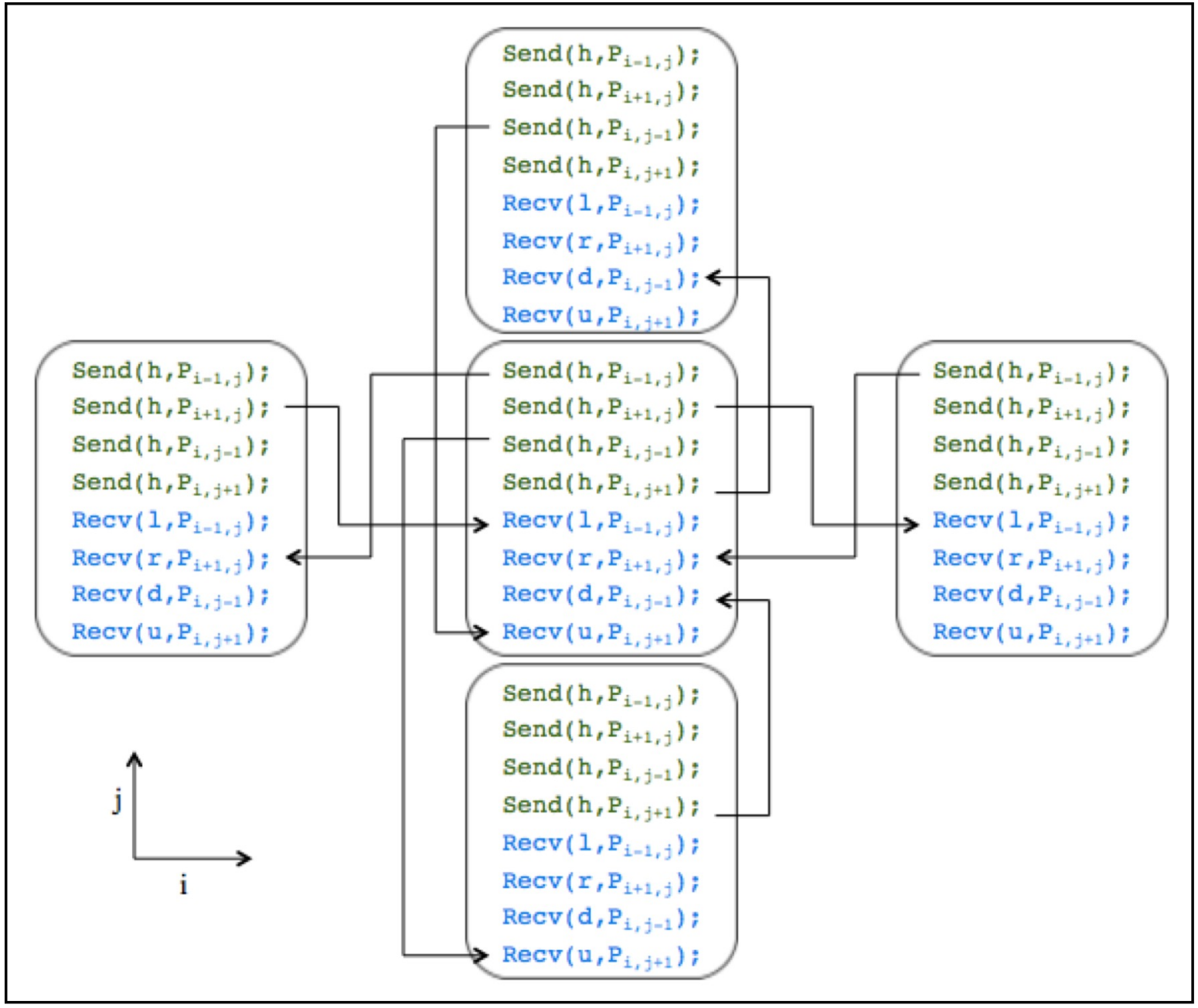
Local barrier

```
1. Parrallel_heat_distribution () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       forall (i=1; i<n; i++)
5.         forall (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                               h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

- Message-passing

  Important to use **send()**s that do not block while waiting for the **recv()**s; otherwise the processes would deadlock, each waiting for a **recv()** before moving on - **recv()**s must be synchronous and wait for the **send()**s.

Message passing for heat distribution problem (1)



Message passing for heat distribution problem (2)

- Master/Slave

$$h_{i,j} \leftrightarrow h$$

$$h_{i-1,j} \leftrightarrow l$$

$$h_{i+1,j} \leftrightarrow r$$

$$h_{i,j-1} \leftrightarrow d$$

$$h_{i,j+1} \leftrightarrow u$$

```
1.  k=0; // Bước lặp thứ k
2.  do {
3.    k++;
4.    h = 0.25 * (l + r + d + u);
  // Send() ở chế độ không bị chặn (non-blocking)
5.    Send(&h, P_{i-1,j});
6.    Send(&h, P_{i+1,j});
7.    Send(&h, P_{i,j-1});
8.    Send(&h, P_{i,j+1});
  // Recv() ở chế độ hay đồng bộ(synchronous) bị chặn (blocking)
9.    Recv(&l, P_{i-1,j});
10.   Recv(&r, P_{i+1,j});
11.   Recv(&d, P_{i,j-1});
12.   Recv(&u, P_{i,j+1});
13. } while (!converged(i, j) && (k < Max_loop));
14. Send(&h, &i, &j, &k, P_master);
```

Asynchronous Computations

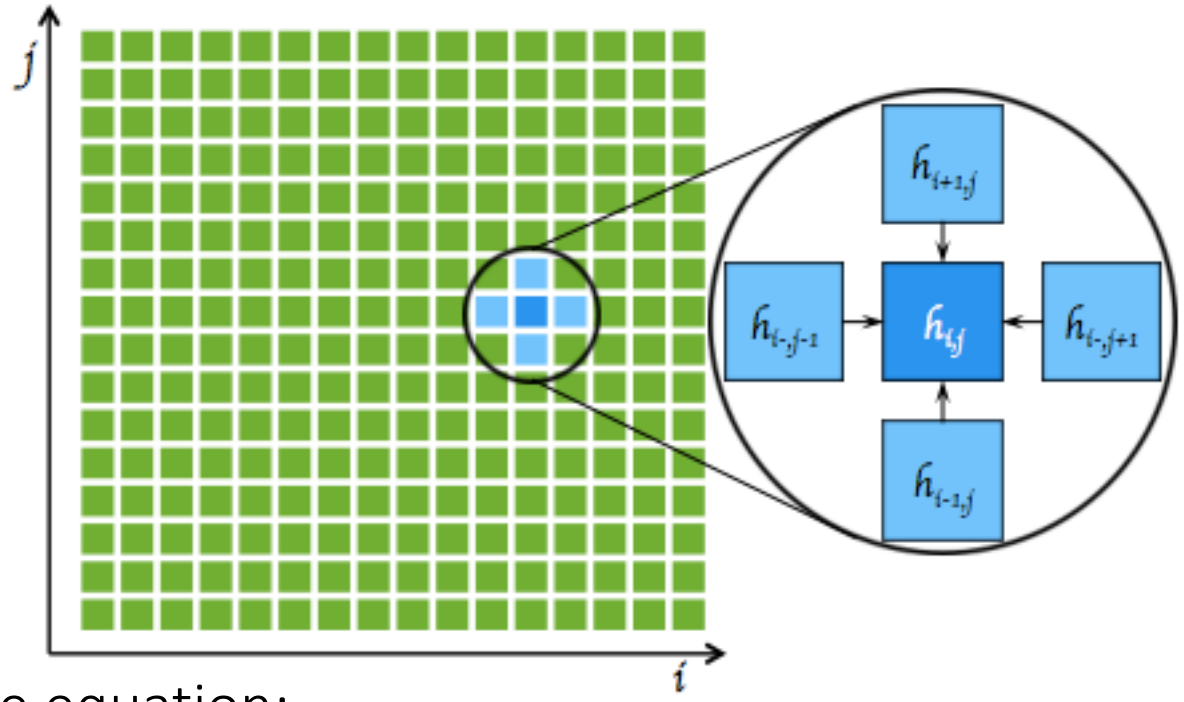
Asynchronous computations

Computations in which individual processes operate without needing to synchronize with other processes.

- Asynchronous computations important because synchronizing processes is an expensive operation which very significantly slows the computation - A major cause for reduced performance of parallel programs is due to the use of synchronization
- Global synchronization is done with barrier routines. Barriers cause processor to wait sometimes needlessly.

Heat distribution problem
(Locally synchronous computation)

- An area has known temperatures along each of its edges
- Find the temperature distribution within
- Divide area into fine mesh of points $h_{i,j}$. Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.



  - Temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

($0 < i < n$, $0 < j < n$) for a fixed number of iterations or until the difference between iterations less than some very small amount.

Sequential algorithms

```
1. Seq_heat_distribution_ver1 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Cập nhật giá trị nhiệt mới tại bước k và h[i][j]
7.       for (i=1; i<n; i++)
8.         for (j=1; j<n; j++)
9.           h[i][j] = g[i][j];
   // Kiểm tra điều kiện kết thúc
10.      continue = false;
11.      for (i=1; i<n; i++)
12.        for (j=1; j<n; j++)
13.          if !converged(i, j) {
14.            continue = true;
15.            break;
16.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
17.   } while ((continue == true) && (k < (Max_loop-1)))
18. }
```

```
1. Seq_heat_distribution_ver2 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

Parallel algorithm

```
  // Lặp đến Max_loop
1.  for (k=0; k<Max_loop; k++) {
2.     h = 0.25 * (l + r + d + u);
  // Send() ở chế độ
  // không bị chặn (non-blocking)
3.     Send(&h, P_{i-1,j});
4.     Send(&h, P_{i+1,j});
5.     Send(&h, P_{i,j-1});
6.     Send(&h, P_{i,j+1});
  // Recv() ở chế độ hay đồng bộ
  // (synchronous) bị chặn (blocking)
7.     Recv(&l, P_{i-1,j});
8.     Recv(&r, P_{i+1,j});
9.     Recv(&d, P_{i,j-1});
10.    Recv(&u, P_{i,j+1});
11. }
```

```
1. Parrallel_heat_distribution () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       forall (i=1; i<n; i++)
5.         forall (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                               h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

Local barrier

Overhead → Barrier

The waiting can be reduced by not forcing synchronization at each iteration

Asynchronous computations

- First section of code computing the next iteration values based on the immediate previous iteration values is traditional Jacobi iteration method
- Suppose however, processes are to continue with the next iteration before other processes have completed
- Then, the processes moving forward would use values computed from not only the previous iteration but maybe from earlier iterations

- Method then becomes an asynchronous iterative method.

Asynchronous iterative method - Convergence

- Mathematical conditions for convergence may be more strict
- Each process may not be allowed to use any previous iteration values if the method is to converge.

Chaotic Relaxation

A form of asynchronous iterative method introduced by Chazan and Miranker (1969) in which the conditions are stated as "there must be a fixed positive integer s such that, in carrying out the evaluation of the i^{th} iterate, a process cannot make use of any value of the components of the j^{th} iterate if $j < i - s$" (Baudet, 1978).

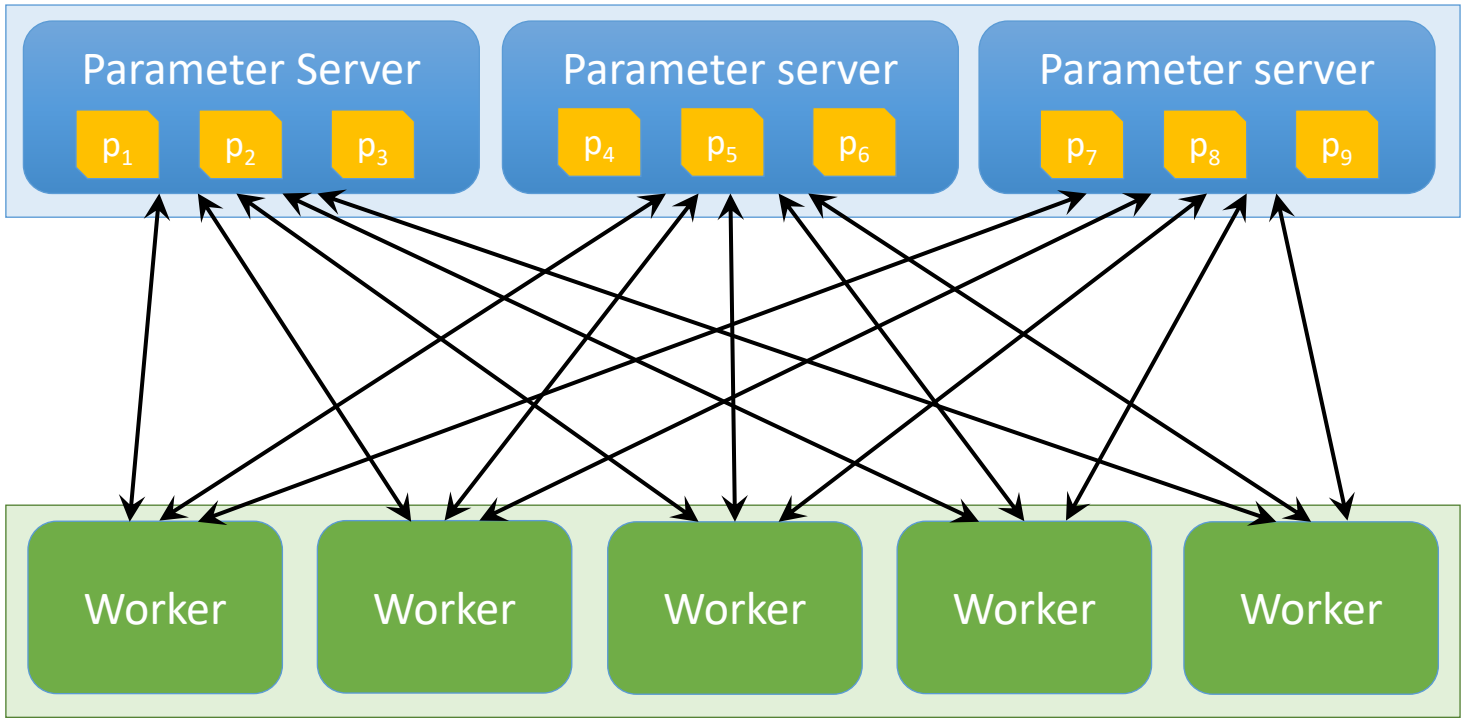
Overall parallel code

- Each process allowed to perform `s` iterations before being synchronized and also to update the array as it goes. At `s` iterations, maximum divergence recorded. Convergence is checked then.
- The actual iteration corresponding to the elements of the array being used at any time may be from an earlier iteration but only up to `s` iterations previously. There may be a mixture of values of different iterations as the array is updated without synchronizing with other processes - truly a chaotic situation.

Parameter Server

Parameter Server (PS)

- Model parameters are stored on PS machines and accessed via key-value interface (distributed shared memory)
- *Extensions*
  - Multiple keys (for a matrix); multiple "channels" (for multiple sparse vectors, multiple clients for same servers, …)
  - Push/pull interface to send/receive most recent copy of (subset of) parameters, blocking is *optional*
  - Can block until push/pulls with clock < $(t - \tau)$ complete



[Smola et al 2010, Ho et al 2013, Li et al 2014]

Machine Learning (ML)

Wide array of problems and algorithms

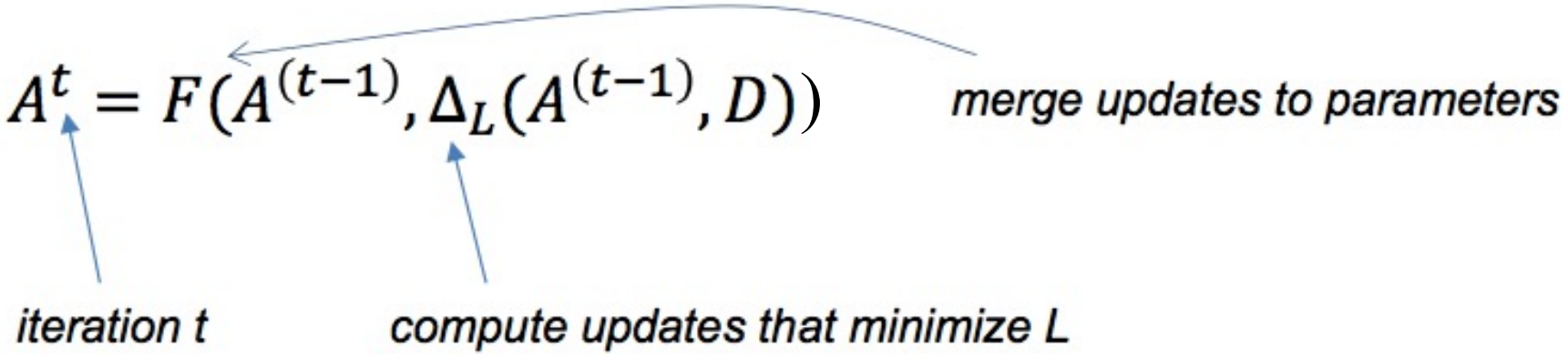
- Classification
  - Given labeled data points, predict label of new data point
- Regression
  - Learn a function from some (x, y) pairs
- Clustering
  - Group data points into “similar” clusters
- Segmentation
  - Partition image into meaningful segments
- Outlier detection

Abstracting ML algorithms

- Can we find commonalities among ML algorithms?
- This would allow finding
  - Common abstractions
  - Systems solutions to efficiently implement these abstractions
- Some common aspects
  - We have a prediction model *A*
  - *A* should optimize some complex *objective function L*
  - ML algorithm does this by iteratively refining *A*

High level view

- Notation
  - *D:* data
  - *A:* model parameters
  - *L:* function to optimize (e.g., minimize loss)
- Goal: Update *A* based on *D* to optimize *L*
- Typical approach: iterative convergence



Distributed Deep Learning Systems (DDLS)

DDLSs train deep neural network models by utilizing the distributed resources of a cluster

- The massive parallel processing power of graphics processing units (GPUs) has been largely responsible for the recent successes in training deep learning models
- Increasingly larger and more complex deep learning models are necessary
- The disruptive trend towards big data has led to an explosion in the size and availability of training datasets for machine learning tasks
  - Training such models on large datasets to convergence can easily take weeks or even months on a single GPU
- Effective remedy to this problem is to utilize multiple GPUs to speed up training
- Scale-up approaches rely on tight hardware integration to improve the data throughput
  - These solutions are effective, but costly
  - Furthermore, technological and economic constraints impose tight limitations on *scaling up*
- DDLS aim at *scaling out* to train large models using the combined resources of clusters of independent machines

Distributed SGD algorithm: all-reduce

- SGD (Stochastic Gradient Descend)

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{i_{b,t}}(w_t),$$

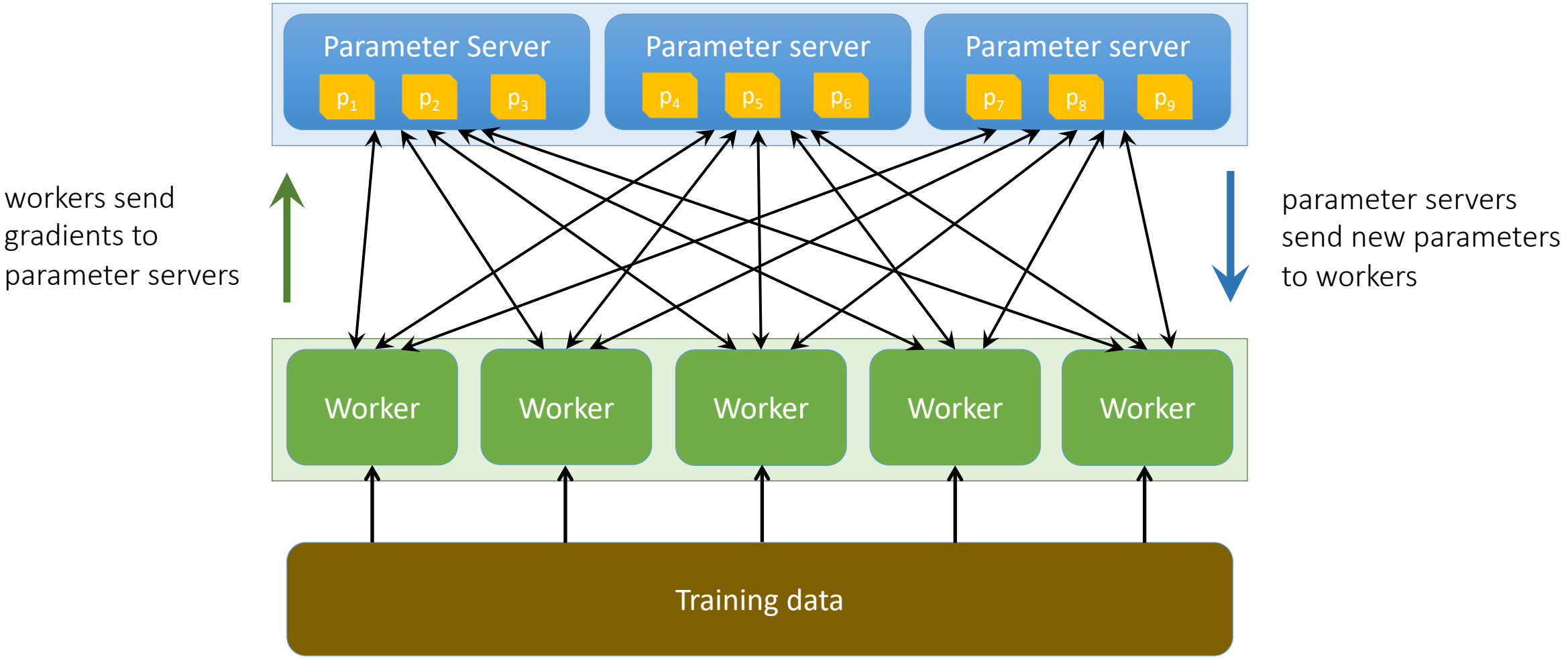
- M machines/mini-batches: B = M.B'

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_t)$$

Algorithm 1 Distributed SGD with All-Reduce

input: loss function examples $f_1, f_2, \ldots$, number of machines M, per-machine minibatch size B'
input: learning rate schedule α_t, initial parameters w_0, number of iterations T
for $m = 1$ **to** M **run in parallel on machine** m
  load w_0 from algorithm inputs
  for $t = 1$ **to** T **do**
    select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
    all-reduce across all workers to compute $G_t = \sum_{m=1}^{M} g_{m,t}$
    update model $w_t \leftarrow w_{t-1} - \frac{\alpha_t}{M} \cdot G_t$
  end for
end parallel for
return w_T (from any machine)

Parameter server (PS)



Algorithm 2 Asynchronous Distributed SGD with the Parameter Server Model

input: loss function examples $f_1, f_2, \ldots$, number of worker machines M, per-machine minibatch size B'
input: learning rate α, initial parameters w_0, number of iterations per worker T
for $m = 1$ **to** M **run in parallel on machine** m
  load $w_{m,0}$ from the parameter server
  for $t = 1$ **to** T **do**
    select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size B'
    compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B} \nabla f_{i_{m,b,t}}(w_{m,t-1})$
    push gradient $g_{m,t}$ to the parameter server
    receive new model $w_{m,t}$ from the parameter server
  end for
end parallel for
run in parallel on param server
  initialize model $w \leftarrow w_0$
  loop
    receive a gradient g from a worker
    update model $w \leftarrow w - \alpha g$
    send w back to the worker
  end loop
end run on param server
return w_T (from any machine)