CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

"Programs must be written for people to read, and only incidentally for machines to execute." — Abelson and Sussman

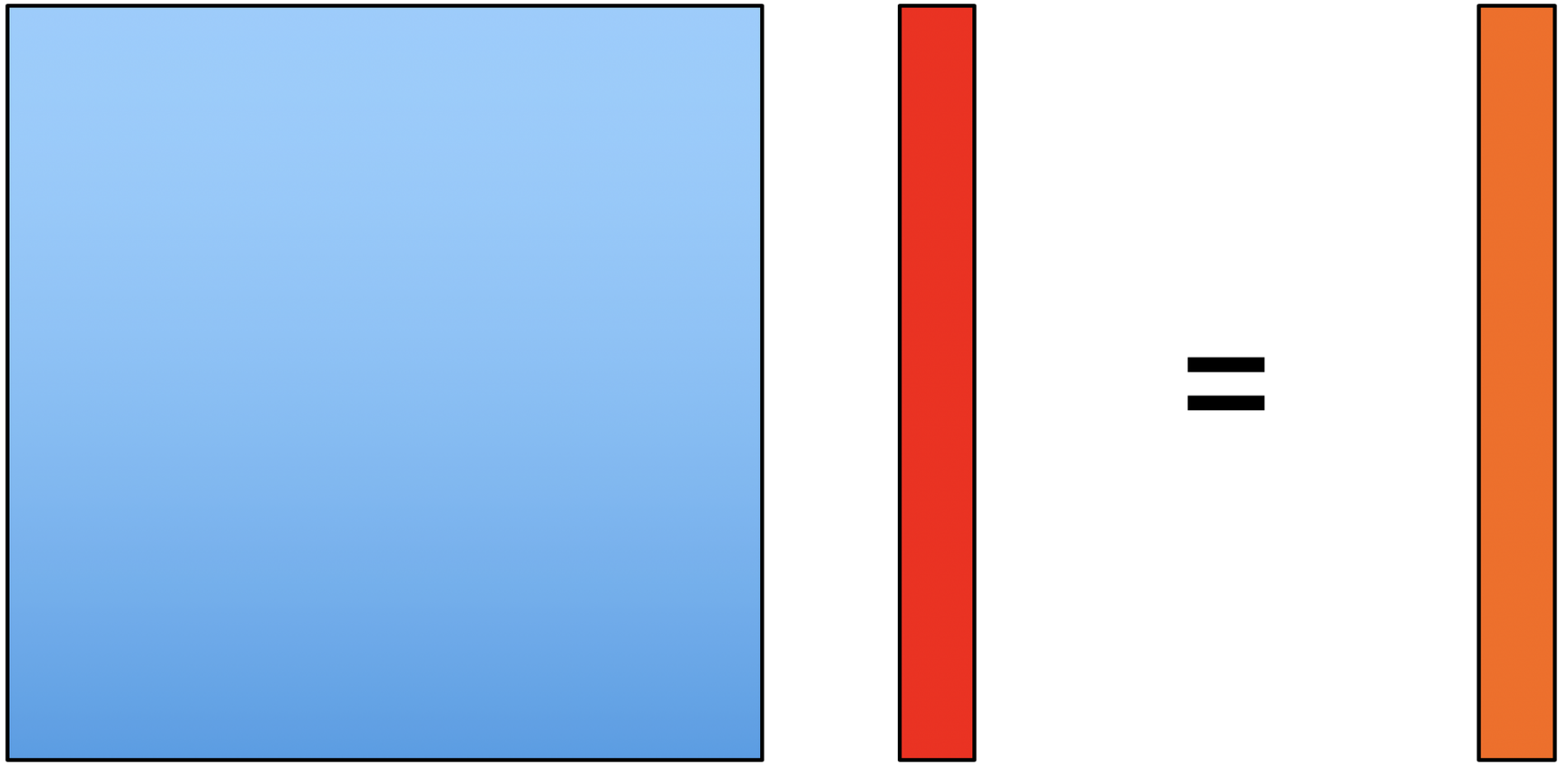
We will illustrate several concepts in distributed memory computing using a linear algebra example, the matrix-vector product.

We will cover two topics:

- understanding and modeling performance
- extending collective communications to groups of processes

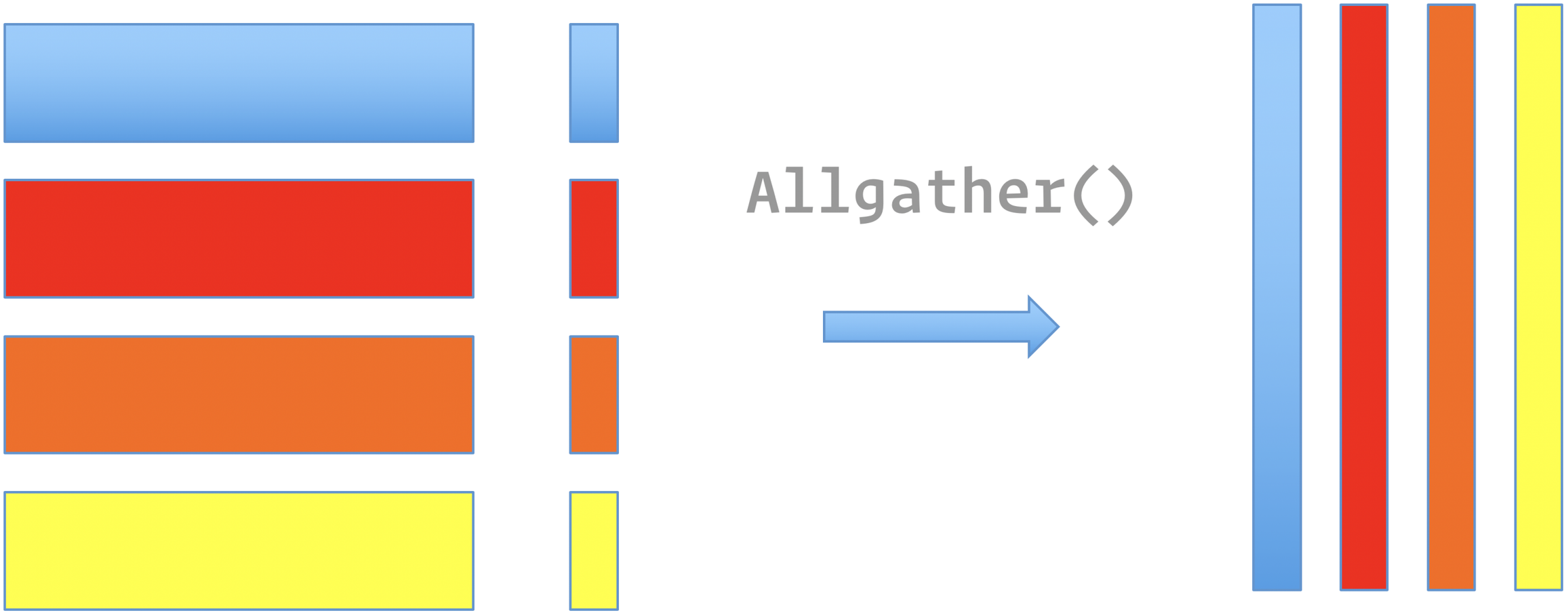
Matrix-vector product

$$x = Ab$$



Matrix A Vector b Vector x

Strategy 1: row partitioning


Allgather()

Step 1: replicate b across all processes

```
MPI_Allgather()
```

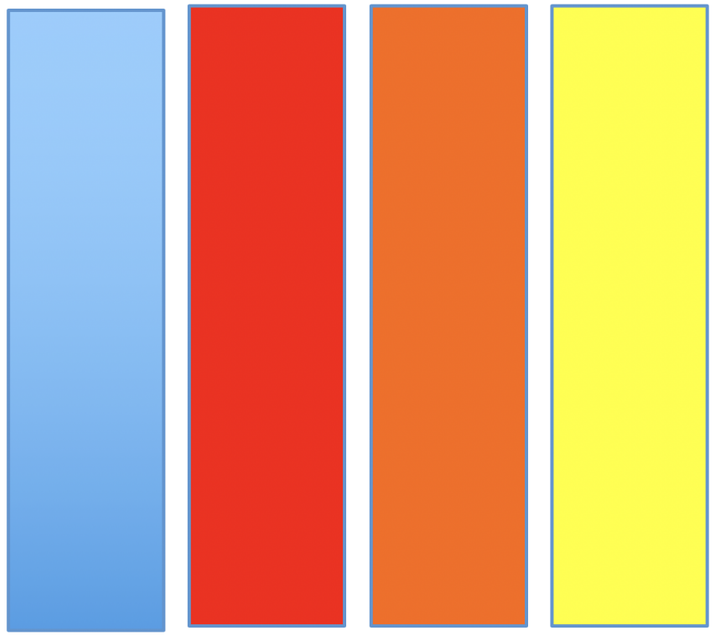
Step 2: local product; no communication

`matvecrow.cpp`

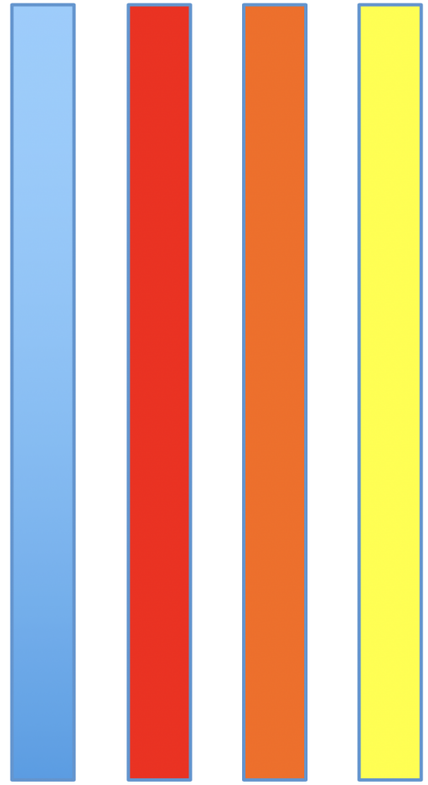
```
// Gather entire vector b on each processor using Allgather
MPI_Allgather(&bloc[0], nlocal, MPI_FLOAT, &b[0], nlocal, MPI_FLOAT,
              MPI_COMM_WORLD);
// sending nlocal and receiving nlocal from any other process

// Perform the matrix-vector multiplication involving the
// locally stored submatrix.
for(int i=0; i<nlocal; i++) {
    for(int j=0; j<n; j++) {
        x[i] += a[i*n+j] * b[j];
    }
}
```

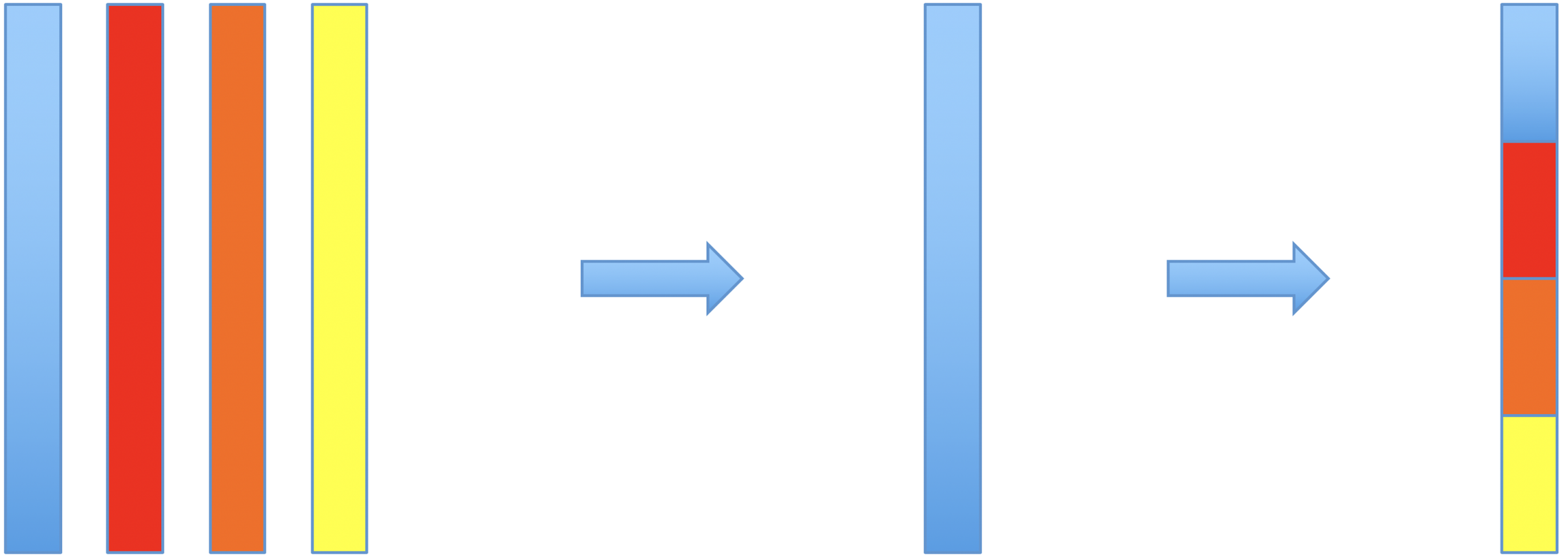
Strategy 2: column partitioning



Partial products



Step 1: calculate partial products with each process



Step 2: reduce all partial results

```
MPI_Reduce()
```

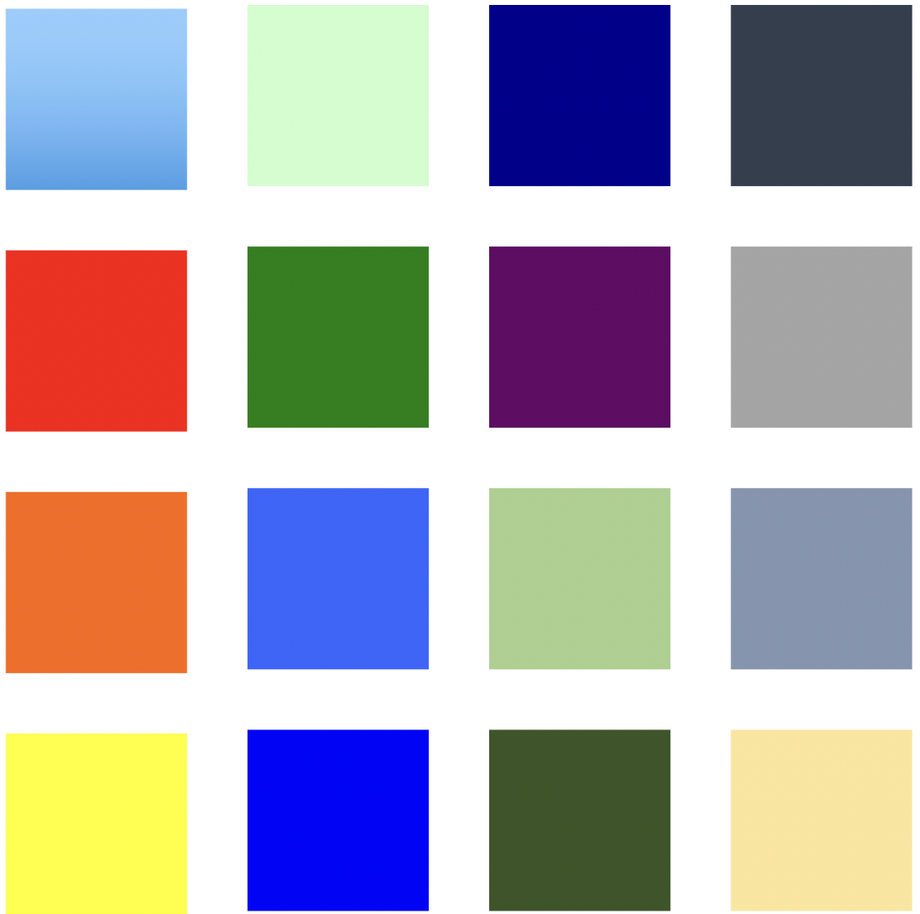
Step 3: send sub-blocks to all processes

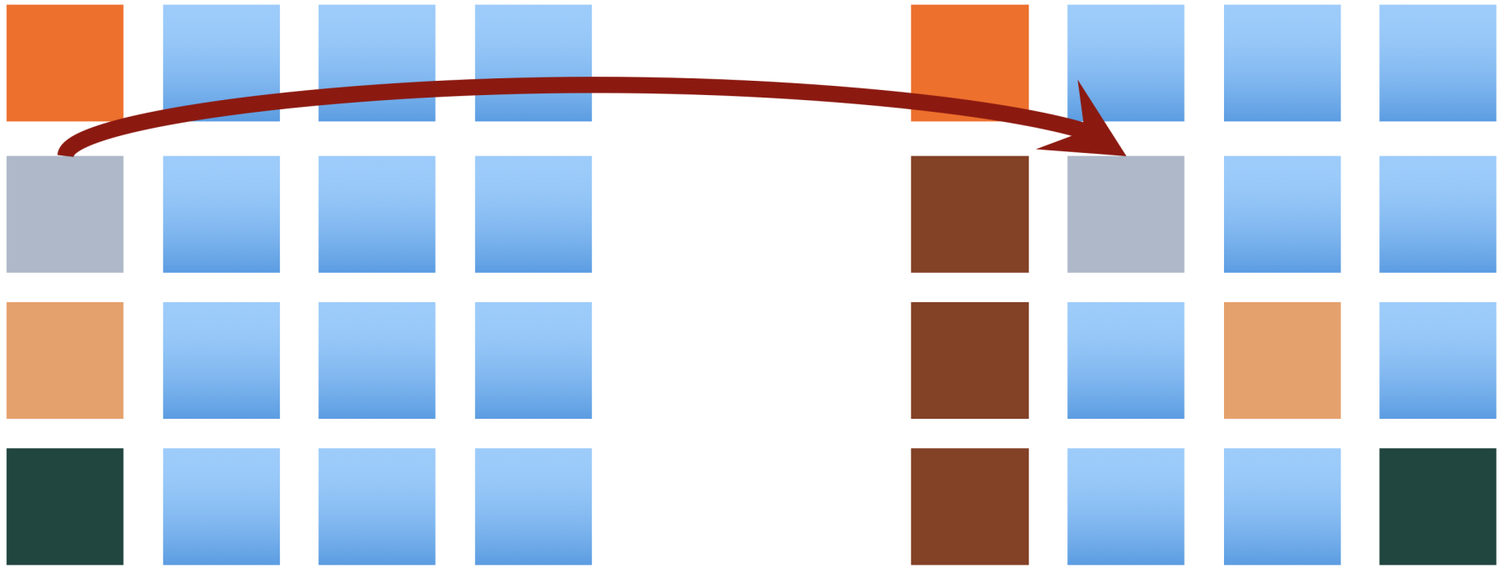
```
MPI_Scatter()
```

Performance is very similar to row partitioning.

If we have many processors, previous approaches lose efficiency.

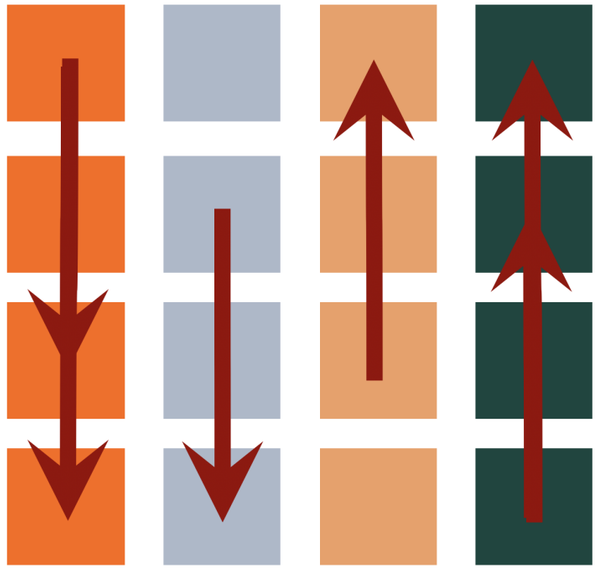
Better approach: 2D block partitioning





First column contains b

Send b to the diagonal processes

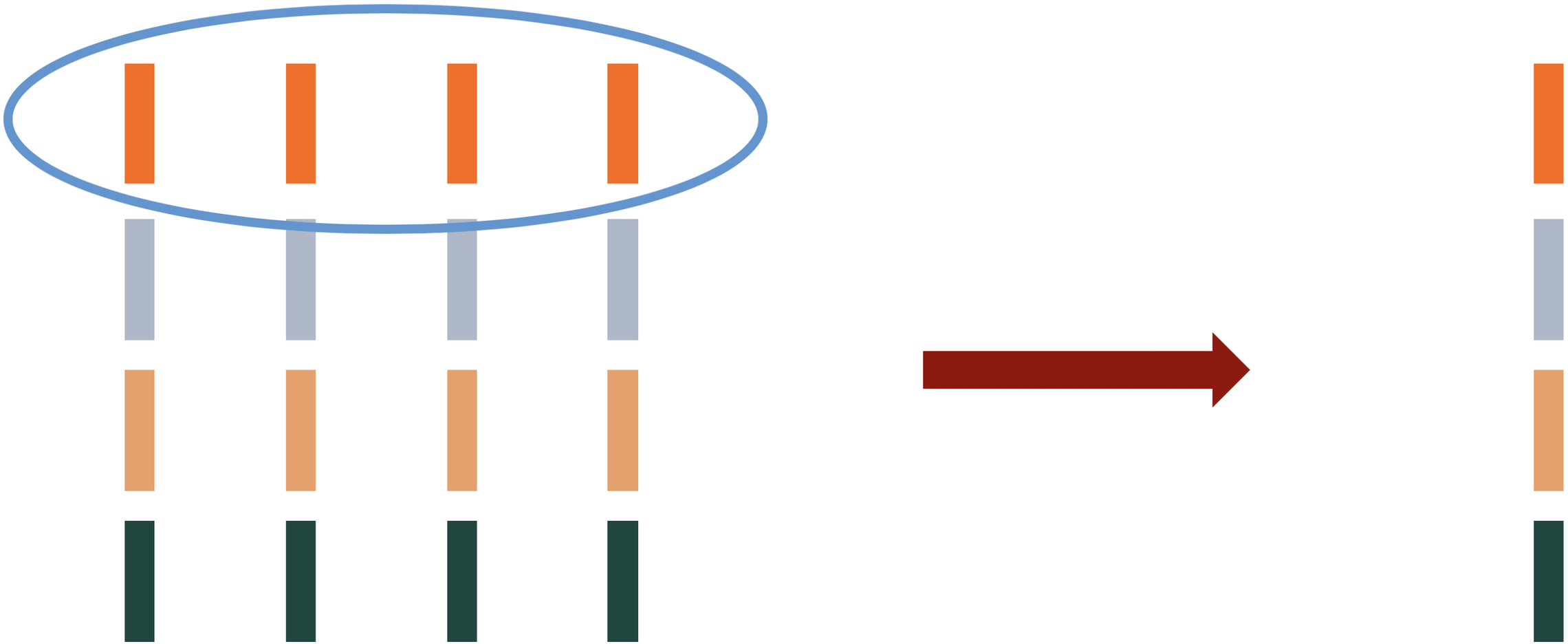


Send b down each column

Step 1: P2P communication

Step 2: broadcast in each column

Step 3: local matrix-vector product



Step 4: reduction across columns

In this approach, we have avoided communications between all processes.

Only subsets communicate.

In addition, we can assign a process per block.

More processes can be used compared to row/column partitioning.

How can we quantify this improvement?

Basic concepts in parallel program efficiency

$T_p(n)$: running time

Matrix of size n and p processes

Breakdown down:

- Computation time
- Communication time
- Idle (waiting on data to continue)

Speedup

$T_1(n)$ execution time in serial

$$S = \frac{T_1(n)}{T_p(n)}$$

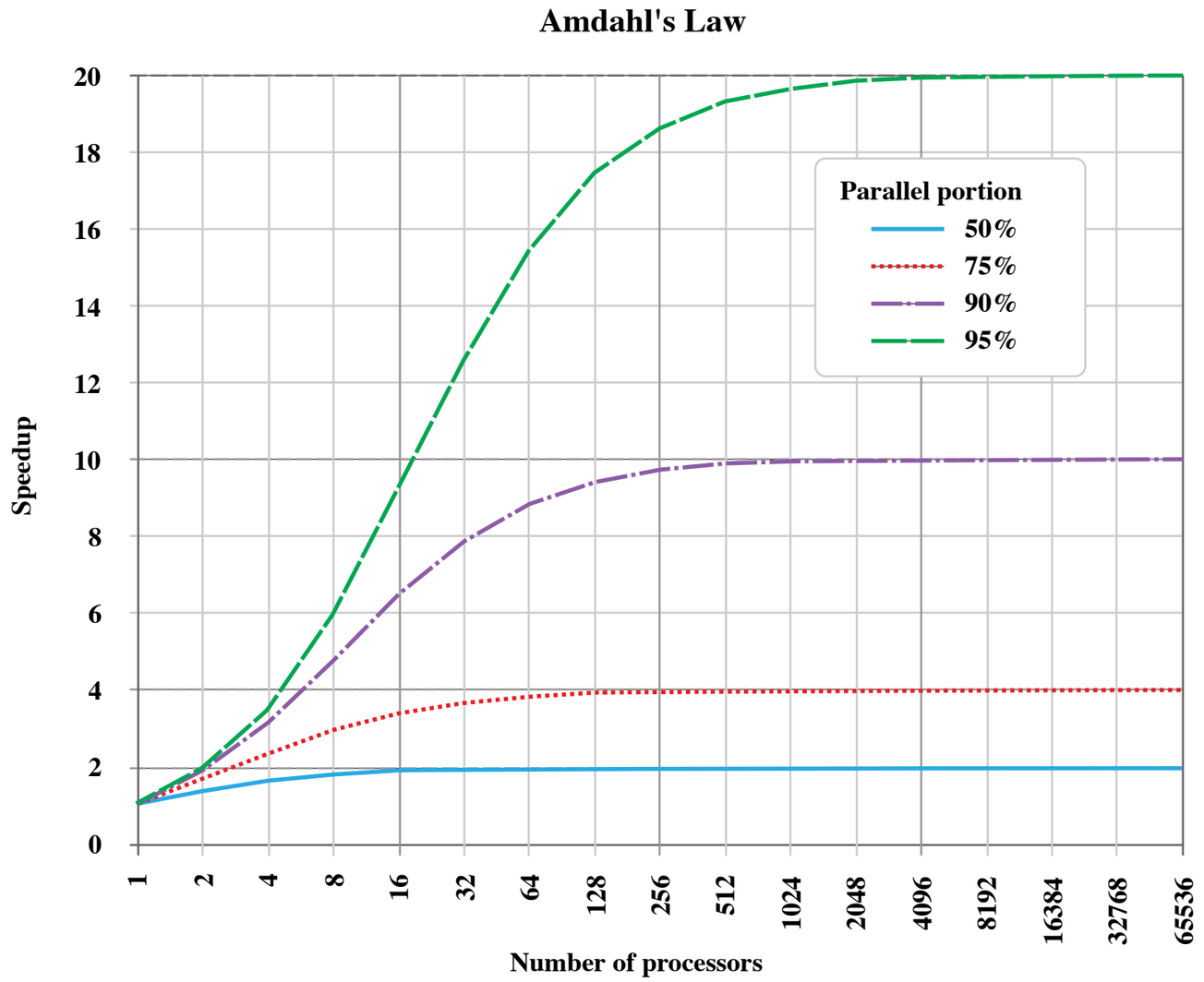
$$S = \frac{T_1(n)}{T_p(n)}$$

Ideally: $S \sim p$

Amdahl's law

$$S_p(n) \sim \frac{T_1(n)}{fT_1(n) + (1-f)T_1(n)/p}$$

$$S_p(n) \sim \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

This law is not perfect

- Decomposition into completely serial and parallelizable is simplistic
- More importantly: f is typically a function of n.

That's why there are programs that can scale to very large sizes

Nevertheless, it contains a key lesson.

As you add more processes to your computation, the serial parts of the algorithm become dominant.

As $p \uparrow$, more and more parts of the program need to be parallelized.

Difficulty ↑

Gustafson's law

The reasoning is different.

Assume that we have access to a larger computer with more computing resources.

We are likely to try to solve a **larger** problem on that computer.

For example, we may decide that we can allocate 1 hour to do the calculation and decide on the problem size based on this requirement.

Workload: $fT_1(n) + (1-f)T_1(n)$.

We expect a speed-up of p on the parallel part.

Assume that we increase the problem size by p so that the overall runtime is about the same.

New workload: $fT_1(n) + (1-f)pT_1(n)$.

The parallel runtime is:

$$fT_1(n) + \frac{(1-f)pT_1(n)}{p} = fT_1(n) + (1-f)T_1(n) = T_1(n)$$

This is what we wanted.

We have a computer that goes p times faster and we have assigned p times more work.

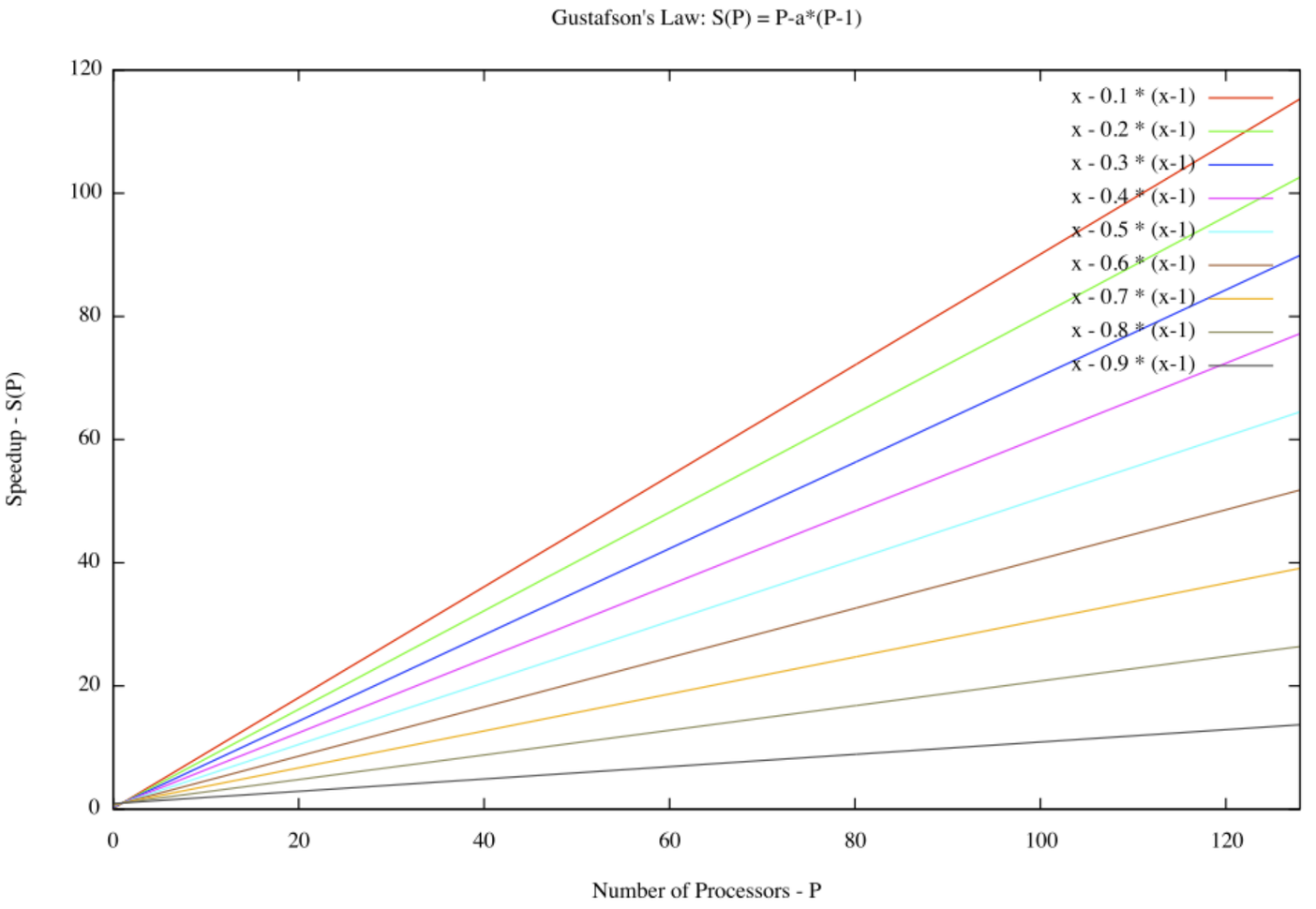
The overall runtime is constant.

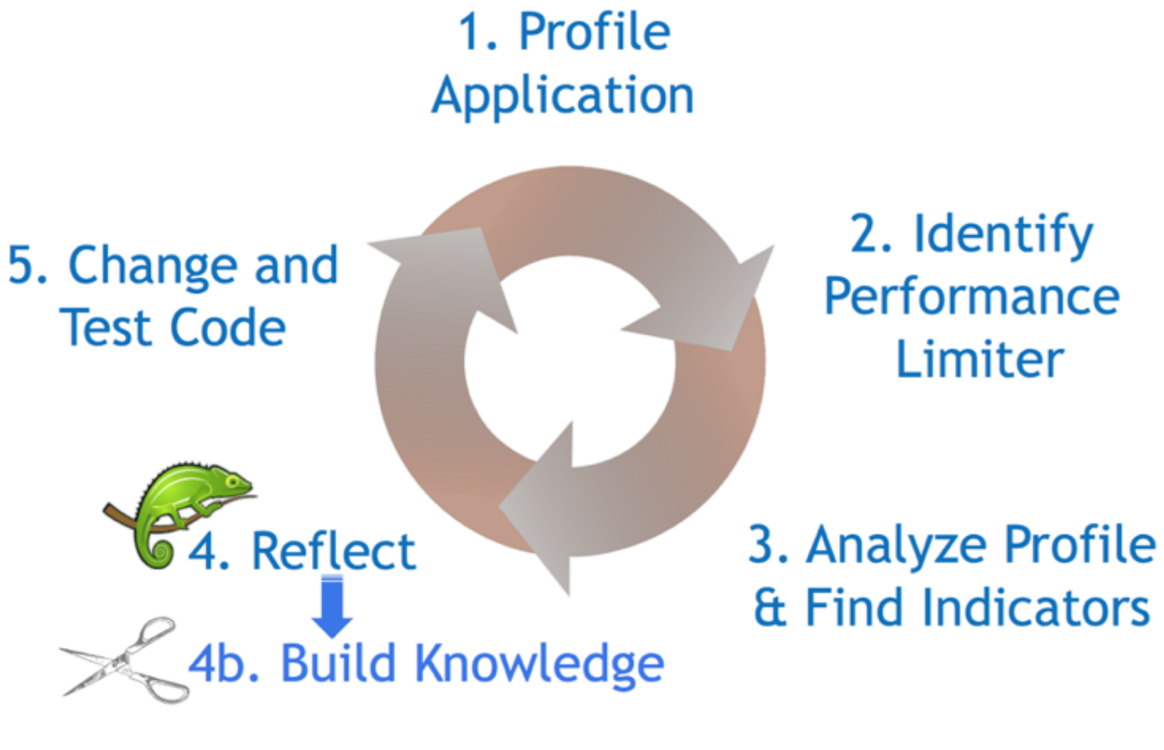
The speed-up is now:

$$S_p(n) \sim \frac{fT_1(n) + (1-f)pT_1(n)}{fT_1(n) + (1-f)T_1(n)}$$

$$S_p(n) \sim f + (1-f)p$$

This is a much more optimistic estimate.


Gustafson's Law: S(P) = P-a*(P-1)
Speedup - S(P)
Number of Processors - P
0
20
40
60
80
100
120
x - 0.1 * (x-1)
x - 0.2 * (x-1)
x - 0.3 * (x-1)
x - 0.4 * (x-1)
x - 0.5 * (x-1)
x - 0.6 * (x-1)
x - 0.7 * (x-1)
x - 0.8 * (x-1)
x - 0.9 * (x-1)


1. Profile Application
2. Identify Performance Limiter
3. Analyze Profile & Find Indicators
4. Reflect
4b. Build Knowledge
5. Change and Test Code

Speedup is difficult to visualize.

Expected to increase like p.

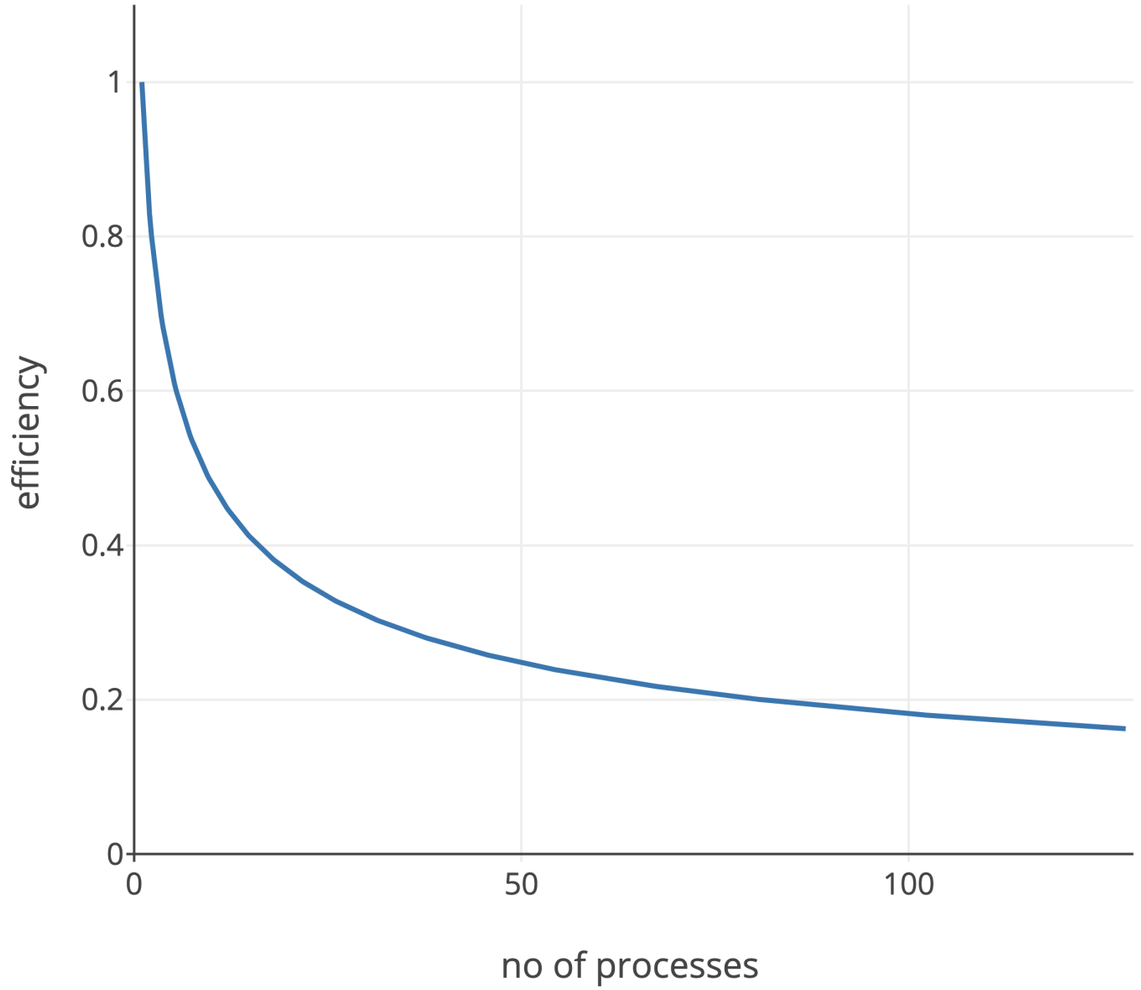
Better is to plot the efficiency.

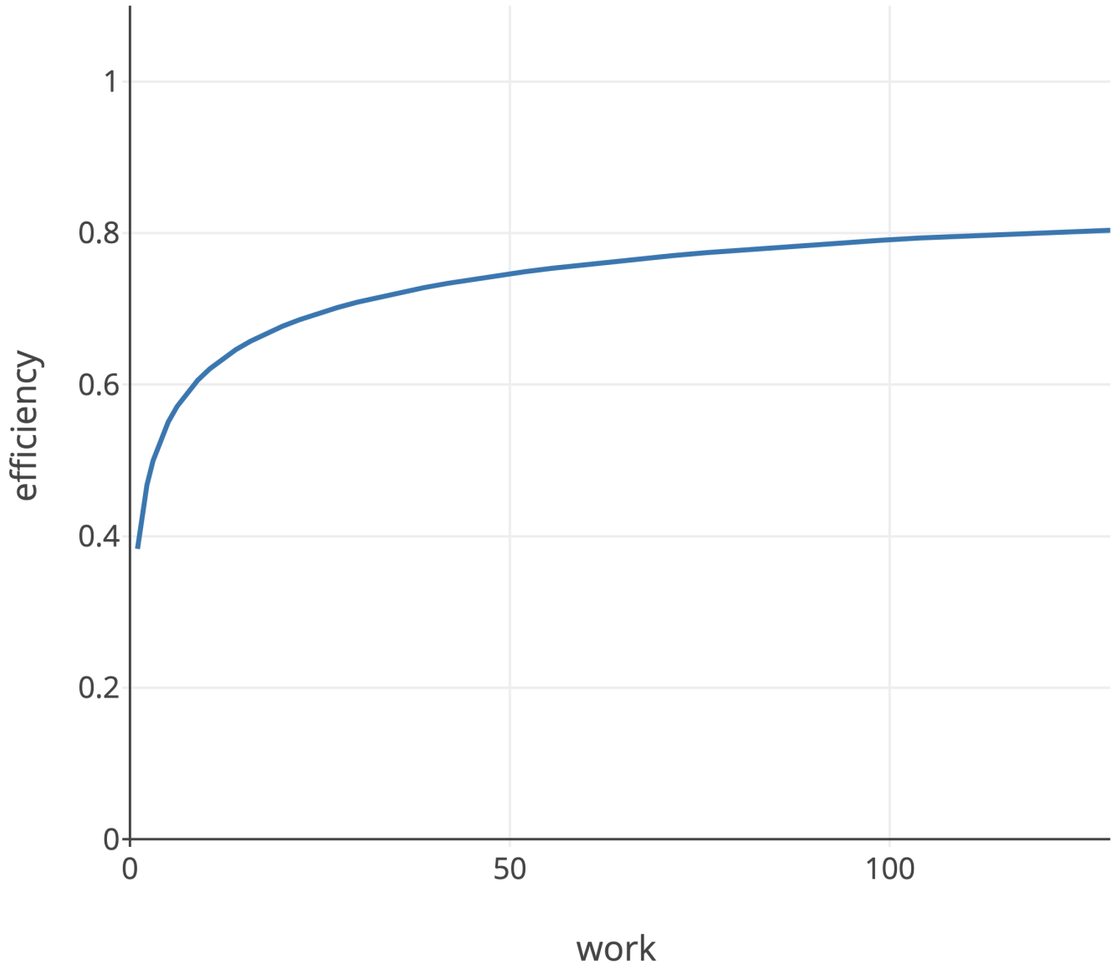
Speedup divided by p:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_1(n)}{pT_p(n)}$$

Ideally, efficiency remains constant as p increases.

Typical trends


efficiency
1
0.8
0.6
0.4
0.2
0
0
50
100
no of processes


1
0.8
0.6
0.4
0.2
0
efficiency
0
50
100
work

Let's apply these ideas to matrix-vector multiplications and see which algorithm is best.

We need to estimate the running time of communication.

Collective communication

Operation	Hypercube time
One-to-all broadcast All-to-one reduction	$\min\{(t_s + t_w m) \log p, 2(t_s \log p + t_w m)\}$
All-to-all broadcast All-to-all reduction	$t_s \log p + t_w m(p-1)$
All-reduce	$\min\{(t_s + t_w m) \log p, 2(t_s \log p + t_w m)\}$
Scatter, Gather	$t_s \log p + t_w m(p-1)$
All-to-all personalized	$(t_s + t_w m)(p-1)$
Circular shift	$t_s + t_w m$

m: size of message

p: number of processes

t_s: latency

t_w: reciprocal bandwidth

All-to-all broadcast: process i has data a_i → process i has data $\cup_j a_j$

All-to-all reduction: process i has data $\cup_j a_{i,j}$ → process i has data $\sum_j a_{j,i}$

All-to-all personalized: process i has data $\cup_j a_{i,j}$ → process i has data $\cup_j a_{j,i}$

Circular shift: process i sends data to process $(i + q)$ mod p.

Application to matrix-vector product

Row partitioning

Serial:

$$T_1(n) = \alpha n^2$$

Parallel: computation + communication

$$T_p(n) = \alpha \frac{n^2}{p} + \beta \ln p + \gamma n$$

$$T_1(n) = \alpha n^2$$

$$pT_p = \alpha n^2 + \beta p \ln p + \gamma p n$$

Efficiency: $E = T_1/(pT_p)$

$$E_p(n) = \frac{1}{1 + (\beta/\alpha) p \ln p / n^2 + (\gamma/\alpha) p/n}$$

Iso-efficiency

How quickly can we increase p such that the efficiency is constant?

$$E_p(n) = \frac{1}{1 + (\beta/\alpha)p \ln p/n^2 + (\gamma/\alpha)p/n}$$

If $p = \Theta(n)$, $E_p(n) =$ constant.

Proof: $p \ln p/n^2 \to 0$, and $p/n =$ constant

Compare with 2D block scheme

Computation

$\alpha \frac{n^2}{p}$

Send b to diagonal

$\beta + \gamma \frac{n}{\sqrt{p}}$

$\sqrt{p}$ because of block partition

Broadcast in each column

$(\beta + \gamma \frac{n}{\sqrt{p}}) \log \sqrt{p}$

Reduction across column

$(\beta + \gamma \frac{n}{\sqrt{p}}) \log \sqrt{p}$

$\log \sqrt{p}$ because of collective communication

Efficiency

$$E_p(n) = \frac{1}{1 + (\beta/\alpha)(p \log p)/n^2 + (\gamma/\alpha)(p^{1/2} \log p)/n}$$

Let's compute the iso-efficiency.

We need to look at each term in the denominator. Each term can either:

- go to 0
- go to a constant
- go to ∞

Assume that $(p \log p)/n^2 \sim$ constant. Then

$$p = \Theta(n^2 / \log n)$$

Second term:

$$\frac{p^{1/2} \log p}{n} \sim \frac{n \log^{1/2} n}{n} \to \infty$$

The efficiency $E_p(n)$ goes to 0.

Our assumption that $(p \log p)/n^2 \sim$ constant is wrong.

Assume that 2nd term $(p^{1/2} \log p)/n \sim$ constant. Then

$$p = \Theta(n^2 / \log^2 n)$$

First term becomes:

$$\frac{p \log p}{n^2} \sim \frac{n^2 / \log n}{n^2} \to 0$$

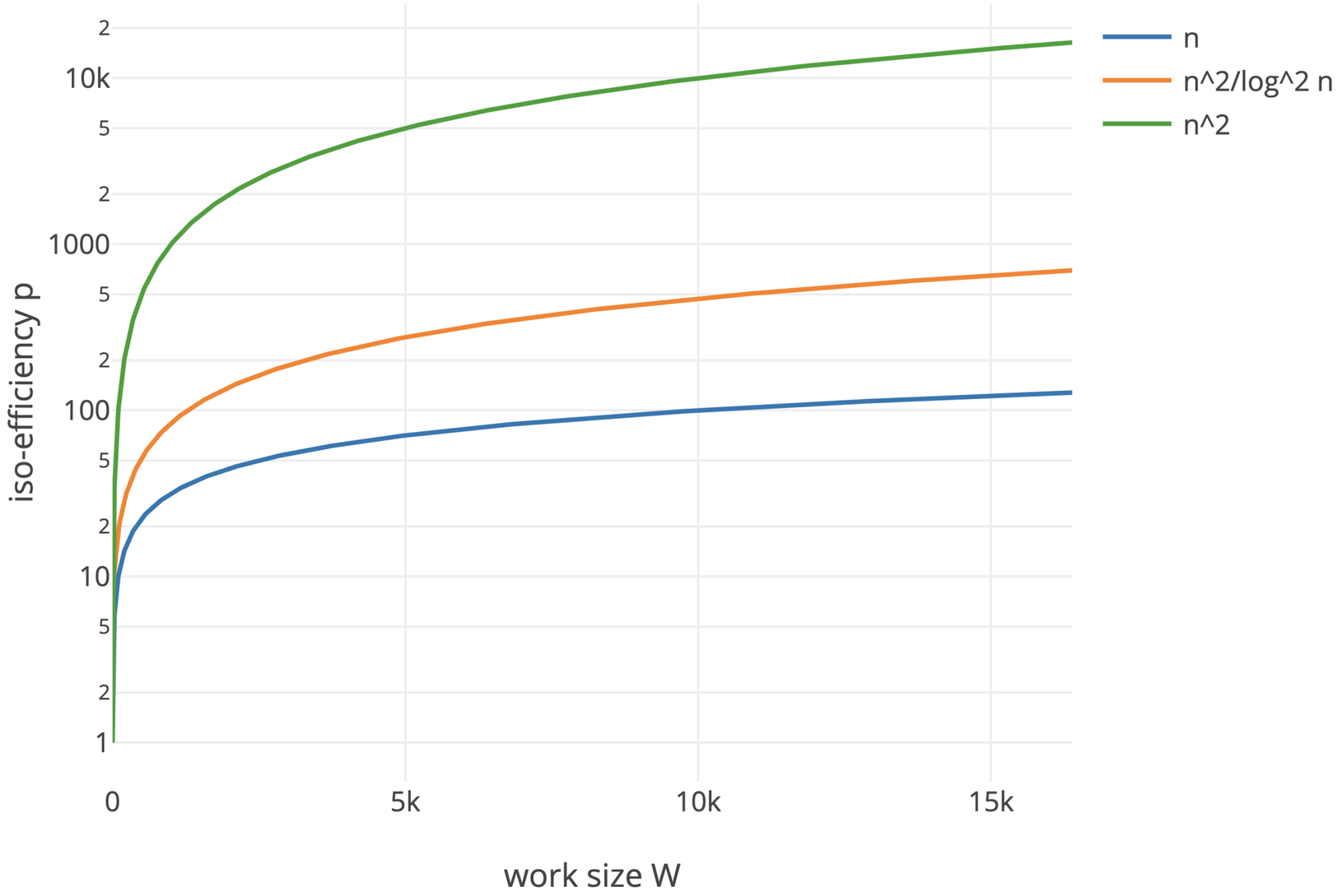
This works. The efficiency converges to a positive constant.

Summary

Row partitioning: $p = \Theta(n)$

2D partitioning: $p = \Theta(n^2 / \log^2 n)$

Which one is better?


n
n^2/log^2 n
n^2
iso-efficiency p
2
10k
5
2
1000
5
2
100
5
2
10
5
2
1
0
5k
10k
15k
work size W

Higher iso-efficiency plot is better.

This means we can maintain the same efficiency but for a larger number of processors.

The code runs faster!

In the matrix-vector algorithm, we did a couple of non-trivial things:

- broadcast data inside a matrix column
- reduce inside a matrix row

Core concept: collective communications with a subset of processors

Groups!

Communicators!

Group

A group of processes used for communication

Communicator

Used to exchange data between processes in the same `group`

MPI provides over 40 routines related to groups, communicators, and virtual topologies!

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Returns group associated with communicator, e.g., `MPI_COMM_WORLD`

```
int MPI_Group_incl(MPI_Group group, int p, int *ranks, MPI_Group *new_group)
```

Creates `new_group` with `p` processes.

`ranks` contains the ranks of processes to appear in `new_group`.

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *new_comm)
```

New communicator based on `group`.

`mpi_group.cpp`

`MPI_Comm_create`

All processes in that group must call `MPI_Comm_create` with the same group as argument.

This means that MPI_Comm_create should be called by the same processes, in the same order.

This implies that the set of groups specified across the processes must be disjoint.

```
$ salloc --partition=CME -N 1 -n 8 mpirun mpi_group
Rank= 0; Group rank= 0; recvbuf= 6
Rank= 1; Group rank= 1; recvbuf= 6
Rank= 2; Group rank= 2; recvbuf= 6
Rank= 3; Group rank= 3; recvbuf= 6
Rank= 4; Group rank= 0; recvbuf= 22
Rank= 5; Group rank= 1; recvbuf= 22
Rank= 6; Group rank= 2; recvbuf= 22
Rank= 7; Group rank= 3; recvbuf= 22
```

```
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
int ranks[2][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}};
int mygroup = (rank < NPROCS / 2) ? 0 : 1;
MPI_Group sub_group;
MPI_Group_incl(world_group, NPROCS / 2, ranks[mygroup], &sub_group);
MPI_Comm sub_group_comm;
MPI_Comm_create(MPI_COMM_WORLD, sub_group, &sub_group_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, sub_group_comm);
```