Improving Input-Step Performance

Experiential Learning for Amdahl's & Gustafson's Laws

Joel C. Adams, PhD

Department of Computer Science

Calvin University

Q: What is the *Input Step*?

A: It's a common issue in 'Big Data' problems…

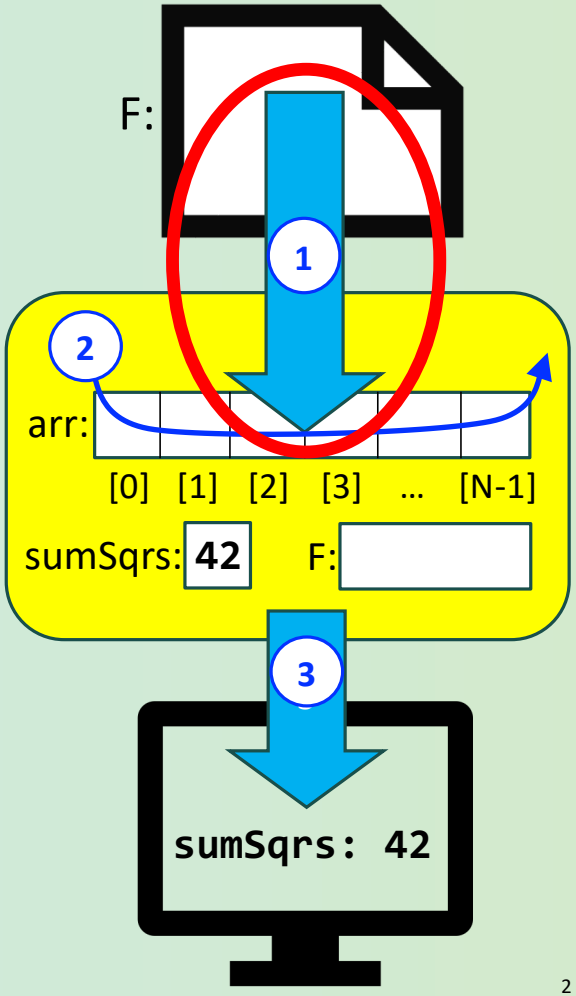
Simple Example: Compute 'Sum of the Squares'

Precondition: Input file **F** contains **N** double values

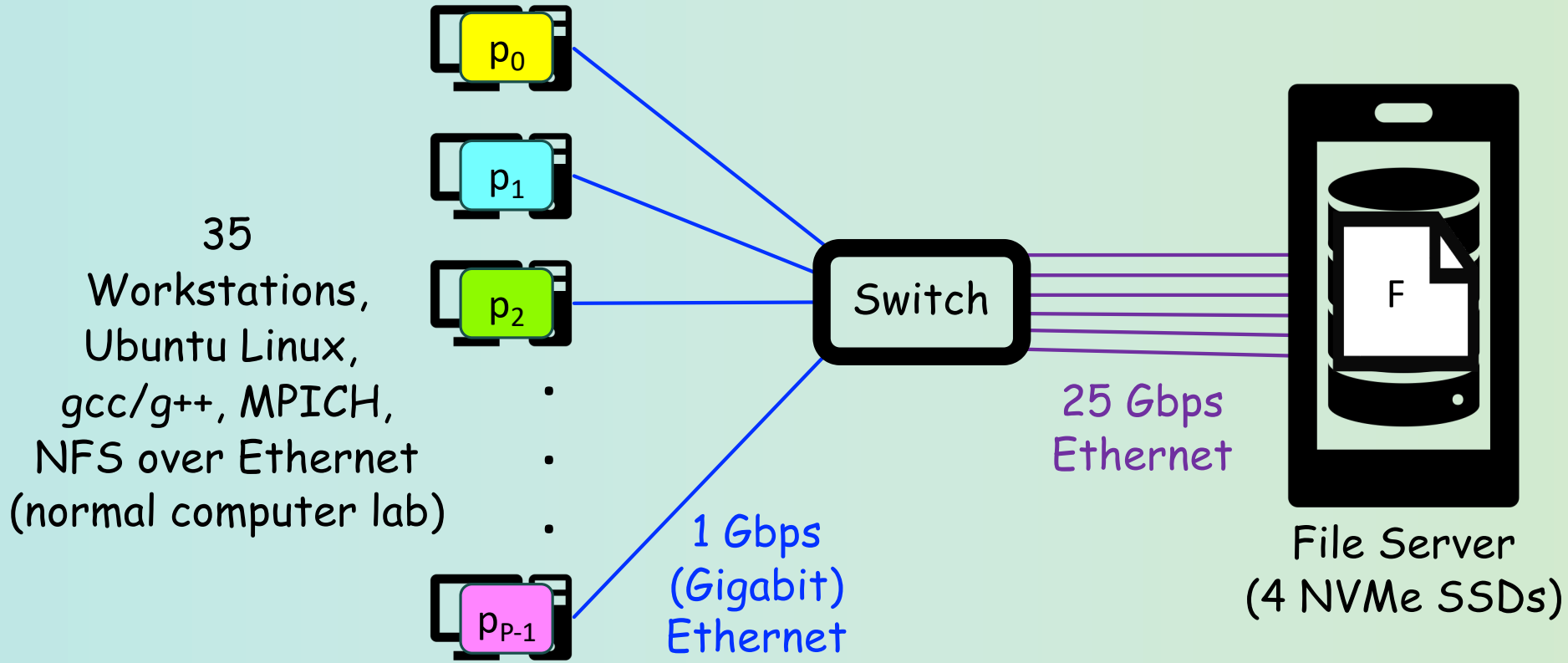
Postcondition: The sum of the squares of the **N** values has been displayed on the screen

Basic algorithm:

1. Read the **N** values from **F** into main memory.
2. Compute **sumSqrs**, the sum of the squares of the **N** values.
3. Output **sumSqrs**.



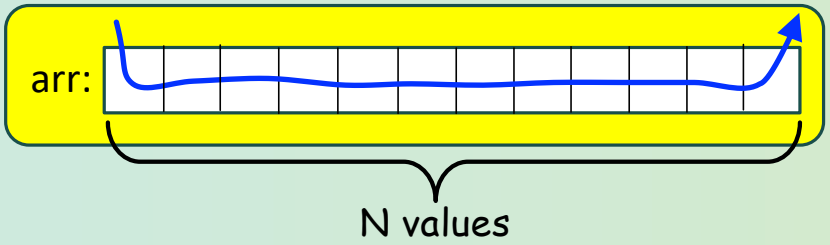
Parallel Hardware: Network of Workstations (NoW)



Step 2 (Computing sumSqrs): Sequential vs. Parallel (4 PEs)

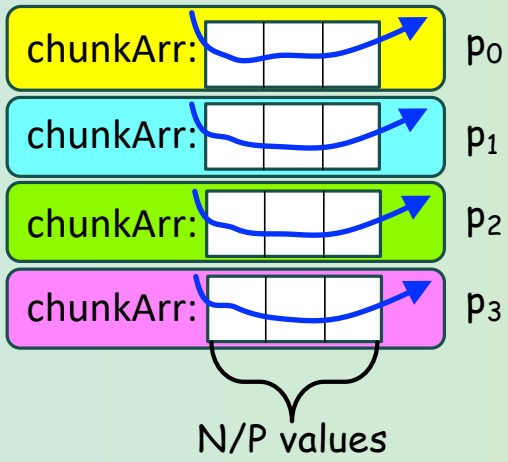
Sequential: If the values are in array **arr**:

```
double sumSqrs = 0.0;
for (int i = 0; i < N; ++i) {
   sumSqrs += (arr[i] * arr[i]);
}
```



Parallel (MPI): If each process's 'chunk' of the values are in **chunkArr**:

```
double sumSqrs = 0.0;
double ssChunk = 0.0;
for (int i = 0; i < chunkSize; ++i) {
   chunkSumSqrs += (chunkArr[i] * chunkArr[i]);
}
MPI_Reduce(&chunkSumSqrs, &sumSqrs, 1,
              MPI_DOUBLE, 0, MPI_COMM_WORLD);
```



Step 1a: Reading N values from F

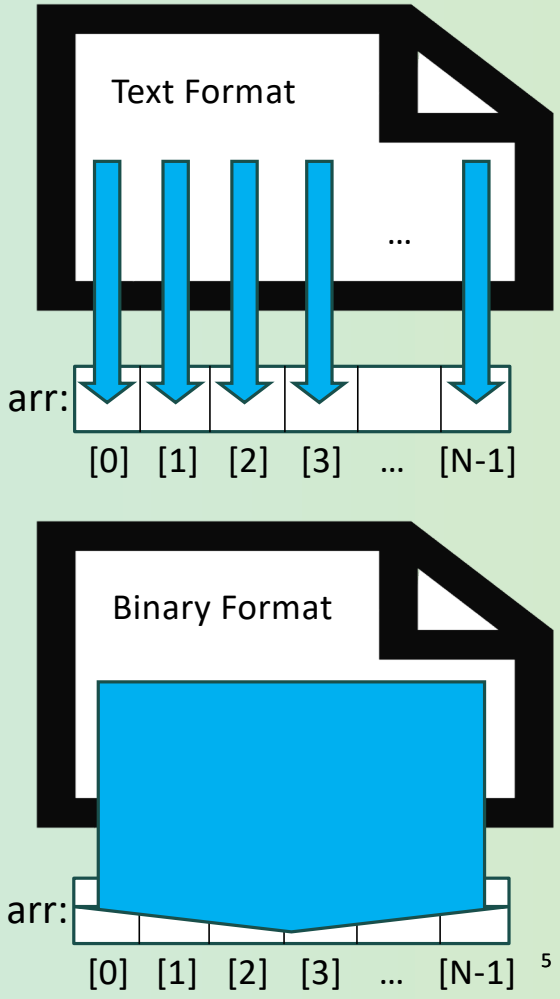
Several possibilities:

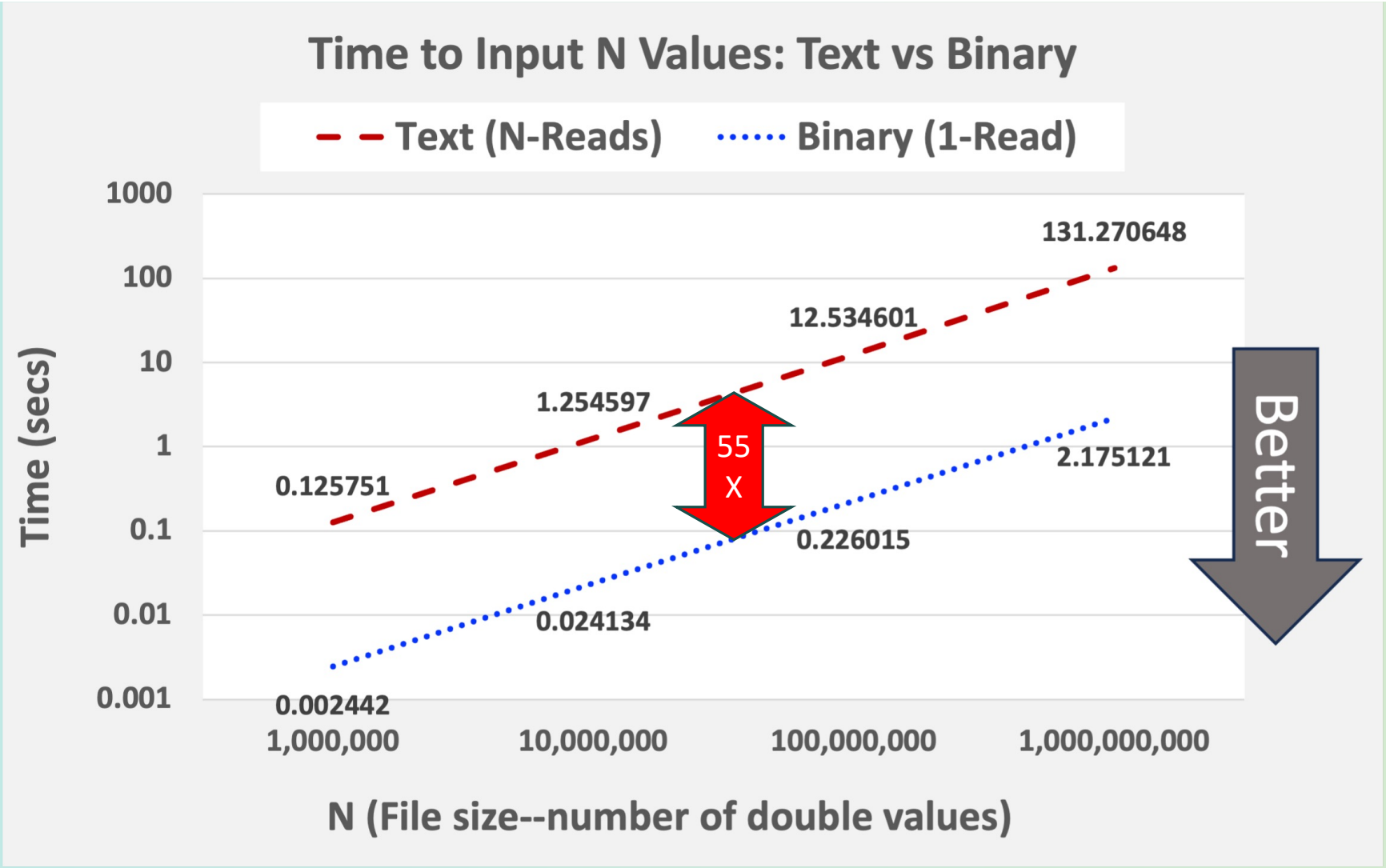
- If F is a text file, read N values *sequentially*
  - Slow: N reads + text-to-binary conversions

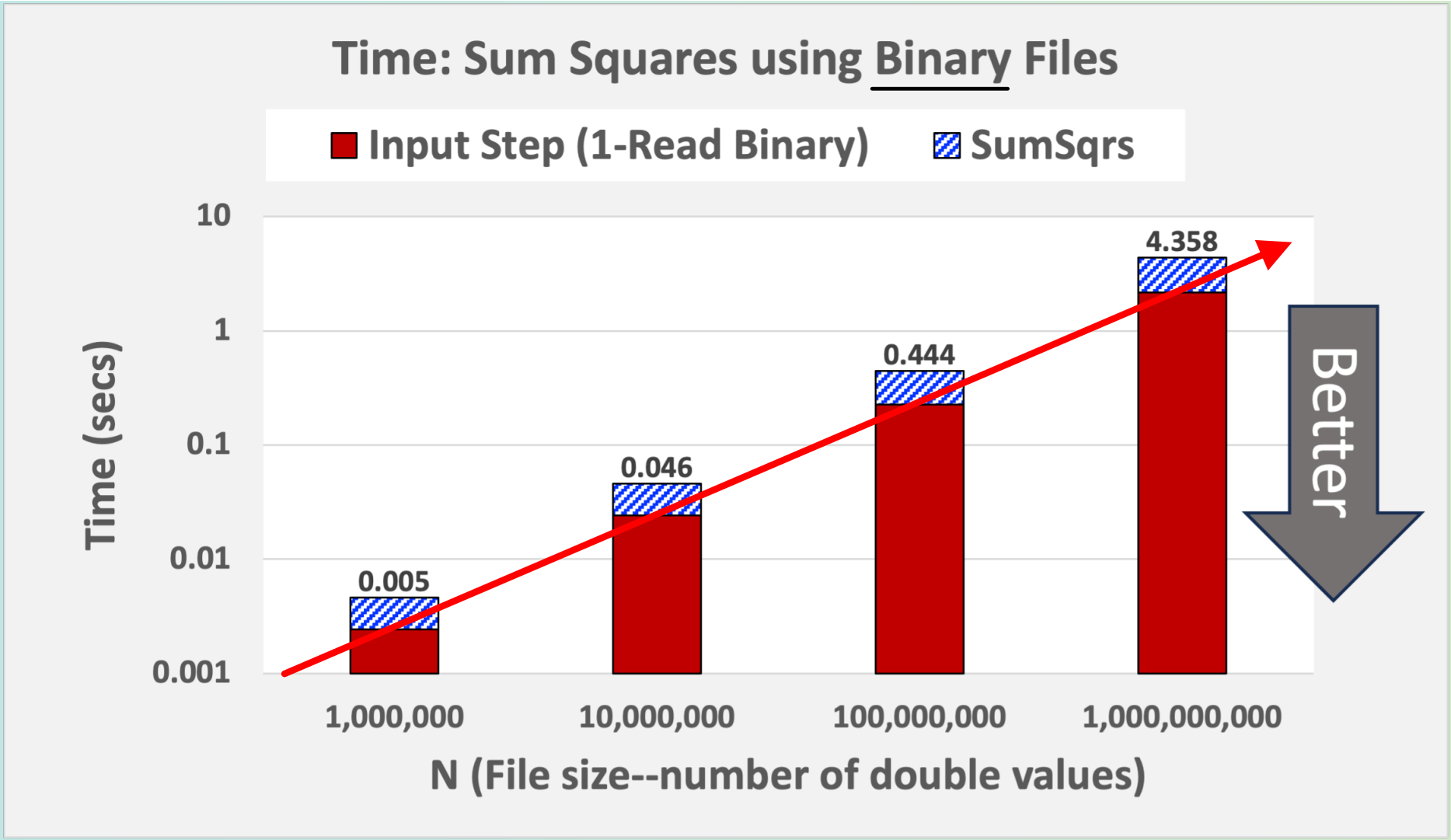
```
FILE *fp = fopen(F, "r");
for (int i = 0; i < N; ++i) {
   fscanf(fp, " %lf", &(arr[i]);
}
```

- If F is a binary file, read N values *sequentially*
  + Fast: 1 read, no conversions needed

```
FILE *fp = fopen(F, "rb");
fread(arr, sizeof(double), N, fp);
```




Time to Input N Values: Text vs Binary
Text (N-Reads)
Binary (1-Read)
1000
100
10
1
0.1
0.01
0.001
Time (secs)
131.270648
12.534601
1.254597
0.125751
55
X
2.175121
0.226015
0.024134
0.002442
Better
1,000,000
10,000,000
100,000,000
1,000,000,000
N (File size--number of double values)


Time: Sum Squares using Binary Files
Input Step (1-Read Binary)
SumSqrs
Time (secs)
10
1
0.1
0.01
0.001
0.005
0.046
0.444
4.358
Better
1,000,000
10,000,000
100,000,000
1,000,000,000
N (File size--number of double values)

Step 1b: Distributing the Data

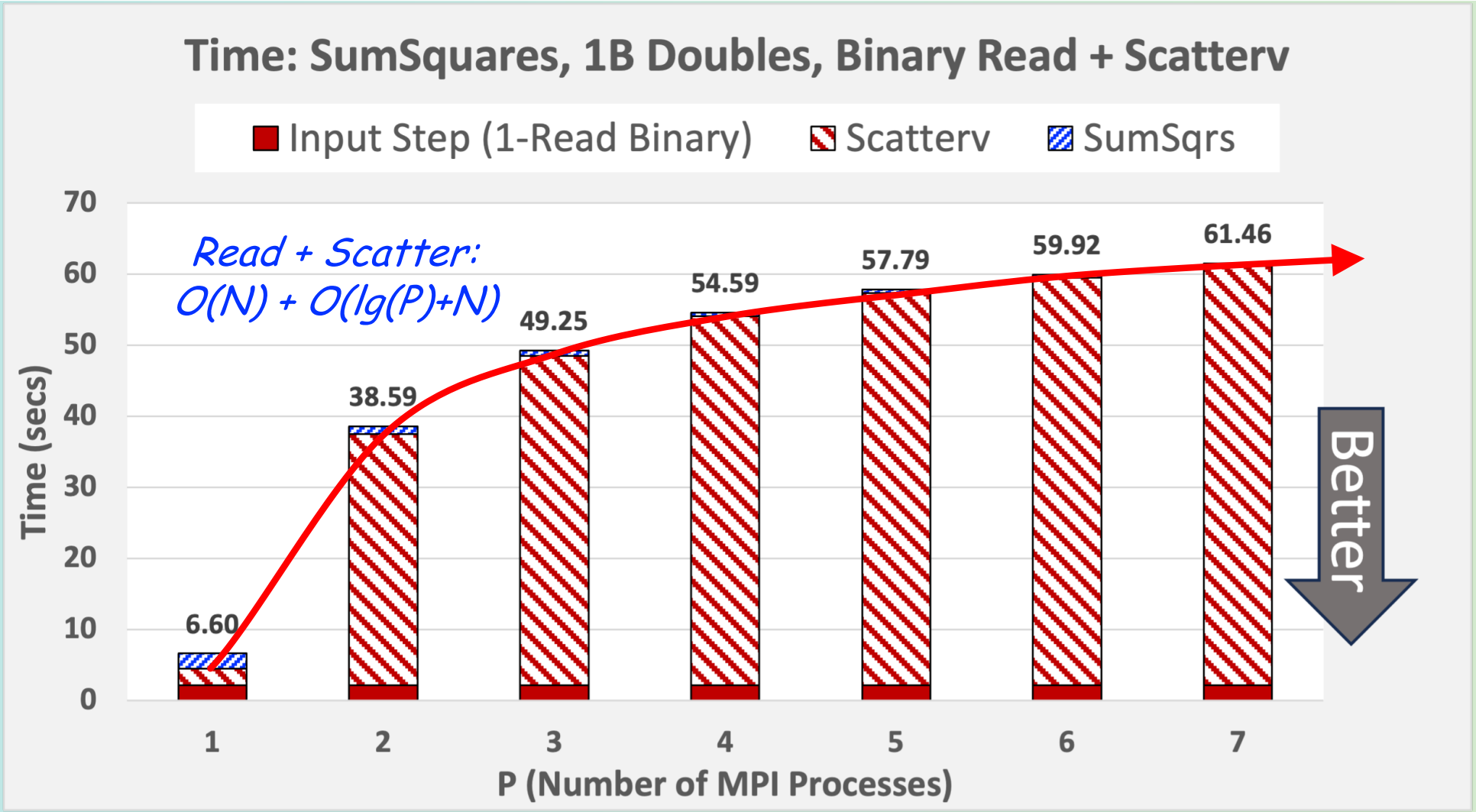
- Distributed processes share no memory

  ⟶ Need a strategy to get the data from file F to the distributed processes

- MPI 1.x strategy:
  a. ‘Master’ process 0 reads the data from F into its array **`arr`**
  b. All processes collectively *Scatter* to get their respective chunks of the data

```
MPI_Scatter(arr, chunkSize, MPI_DOUBLE,
            chunkArr, chunkSize, 0, MPI_COMM_WORLD);
```

  ⟶ Scatter is *slow*, *significantly increasing the time*


Time: SumSquares, 1B Doubles, Binary Read + Scatterv
Input Step (1-Read Binary)
Scatterv
SumSqrs
Read + Scatter:
O(N) + O(lg(P)+N)
Time (secs)
70
60
50
40
30
20
10
0
6.60
38.59
49.25
54.59
57.79
59.92
61.46
1
2
3
4
5
6
7
P (Number of MPI Processes)
Better

Amdahl's and Gustafson's Laws

- **Amdahl's Law**: For a fixed value of N, $Speedup(N, P) \leq 1/seq$, (seq = percentage of program execution that is inherently sequential)

  ⟶ Sequential read: as N increases, seq increases, reducing speedup

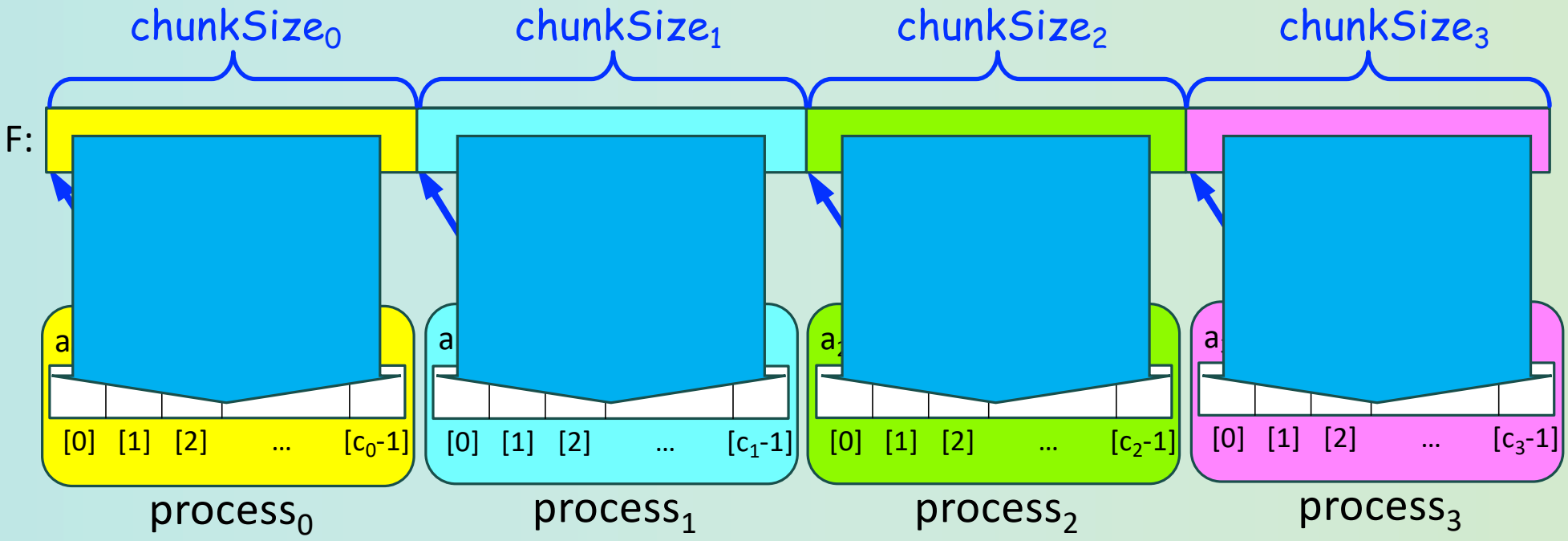
- **Gustafson's Law**: If N can increase, then as $N \rightarrow \infty$, $Speedup(N, P) \rightarrow P$, provided increasing N does not increase seq.

  ⟶ But sequential read takes *O(N),* so increasing *N* increases *seq*

  ⟶ Gustafson's Law is not applicable, so long as we read sequentially!

Bottom Line: Gustafson can only 'rescue' us from Amdahl if we can perform the input step *non-sequentially*…

Visualizing Parallel Input (4 Processes)



This should reduce the input-step time from *O(N) + O(lg(P)+N)* to *O(N/P)*.

MPI-IO (Portable Parallel I/O since MPI 2): Input Step

P MPI processes simultaneously and collectively:

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, F, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &f);
MPI_File_get_size(f, &fSize);
itemSize = sizeof(Item);
N = fSize / itemSize;
getChunkStartAndStop(id, P, N, &start, &stop);
chunkSizeInItems = stop - start;
chunkSizeInBytes = chunkSizeInItems * itemSize;
Item* chunkArr = malloc(chunkSizeInBytes);
chunkByteOffset = start * itemSize;
MPI_File_read_at_all(f, chunkByteOffset,
                     chunkArr, chunkSizeInBytes,
                     MPI_DOUBLE, &status);
MPI_File_close(&f);
```

1a. Open input file *F*

1b. Determine the size and offset of this process's chunk of *F*

1c. Read this process's chunk of *F* into chunk-array

1d. Close *F*

Problem

- Since 1997, I have taught CS 374: *High Performance Computing*, a biannual junior-senior elective course at Calvin University.
- Between 2011 and 2021, I assigned this sum-the-squares problem; students could choose either *read+scatter* or *parallel I/O* (MPI-IO) for the input step; I expected stronger students to choose MPI-IO.
- Out of 120+ students, *zero* chose parallel IO!
- When I asked my 2021 students why, they said that when they were designing their solutions, read+scatter seemed simpler than MPI-IO…

  ⟶ They understood the advantage of parallel I/O, but found MPI-IO intimidating!

Solving the Complexity Problem: Abstraction

Hide the complexity of all this C:

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, F, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &f);
MPI_File_get_size(f, &fSize);
itemSize = sizeof(Item);
N = fSize / itemSize;
getChunkStartAndStop(id, P, N, &start, &stop);
chunkSizeInItems = stop - start;
chunkSizeInBytes = chunkSizeInItems * itemSize;
Item* chunkArr = malloc(chunkSizeInBytes);
chunkByteOffset = start * itemSize;
MPI_File_read_at_all(f, chunkByteOffset,
                     chunkArr, chunkSizeInBytes,
                     MPI_DOUBLE, &status);
MPI_File_close(&f);
```

… within an easy-to-use C++ *parallel input abstraction*:

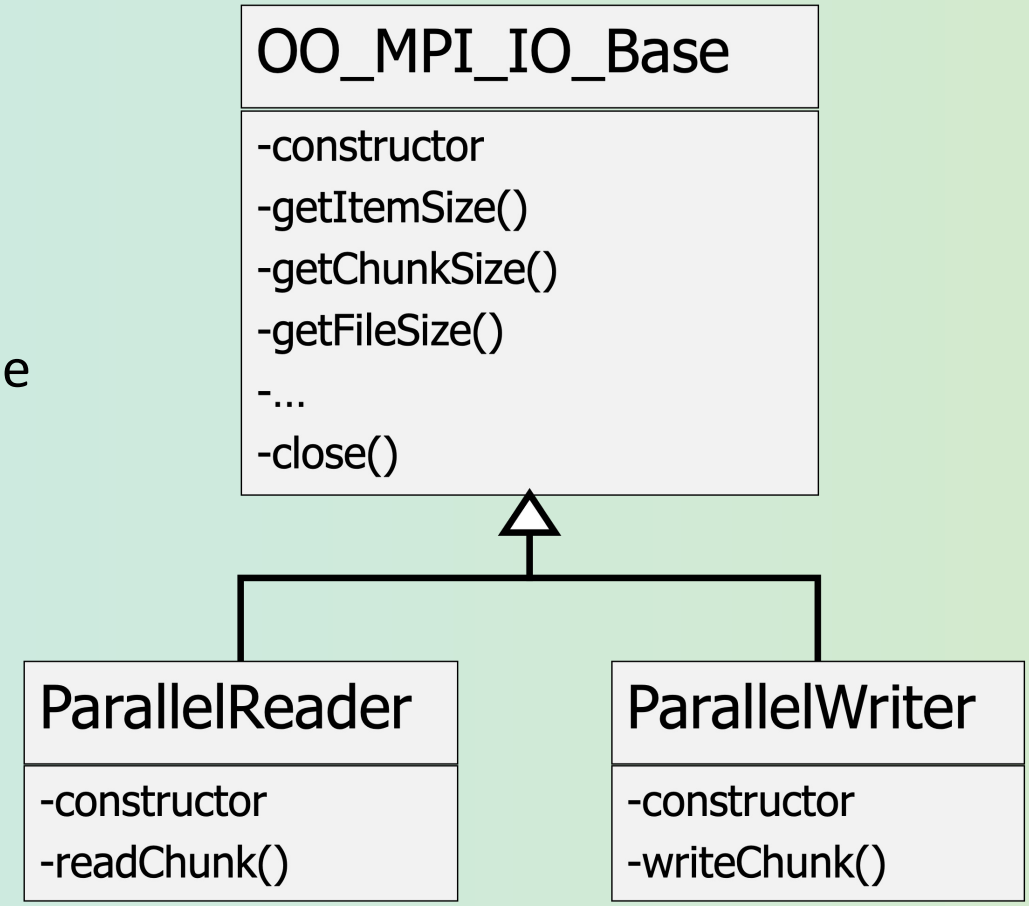
```
ParallelReader<double> reader(F, MPI_DOUBLE, id, P);
std::vector<double> v = reader.readChunk();
reader.close();
```

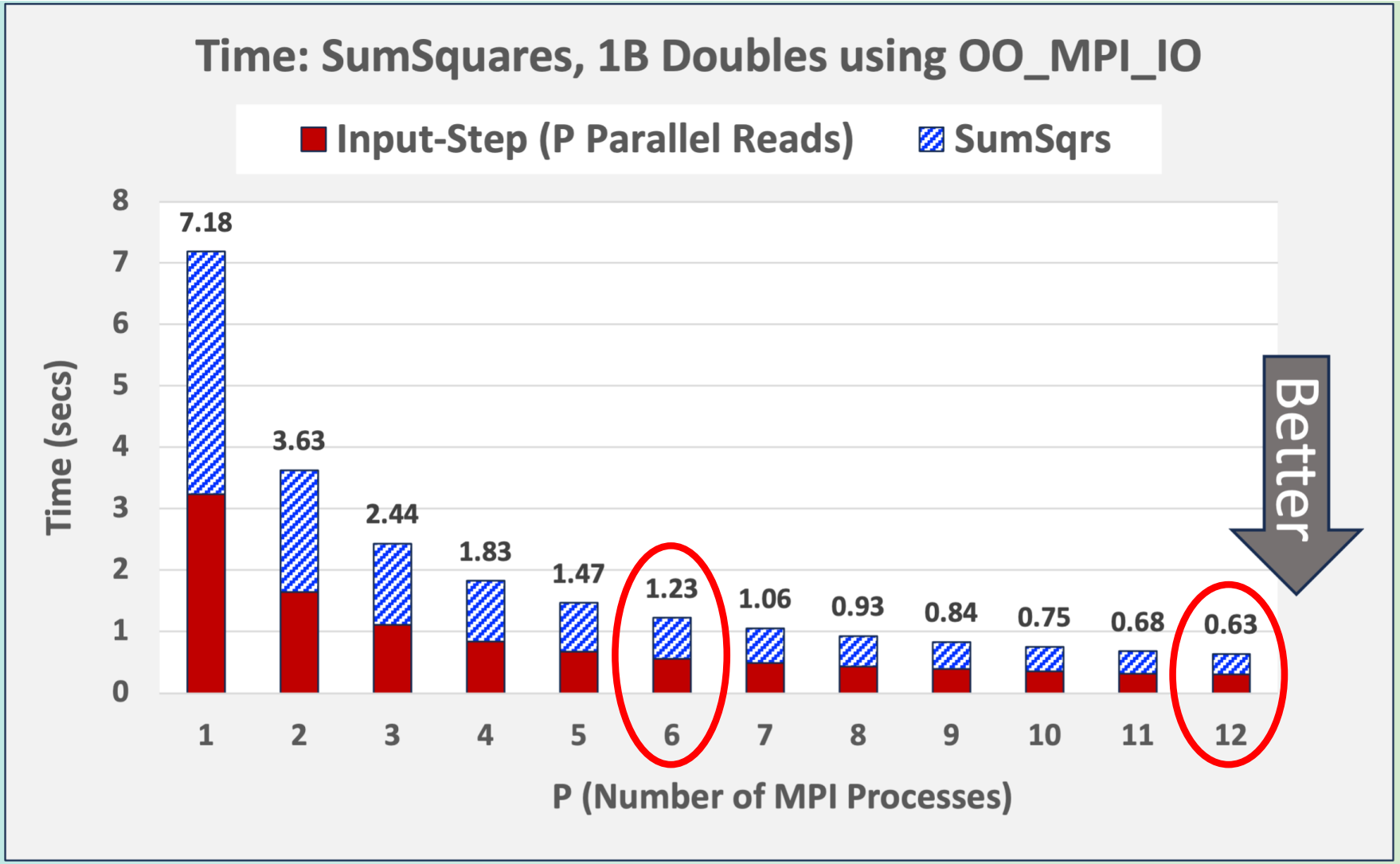
Then create a corresponding *parallel output abstraction*, and use OOD to let them inherit their common functionality…

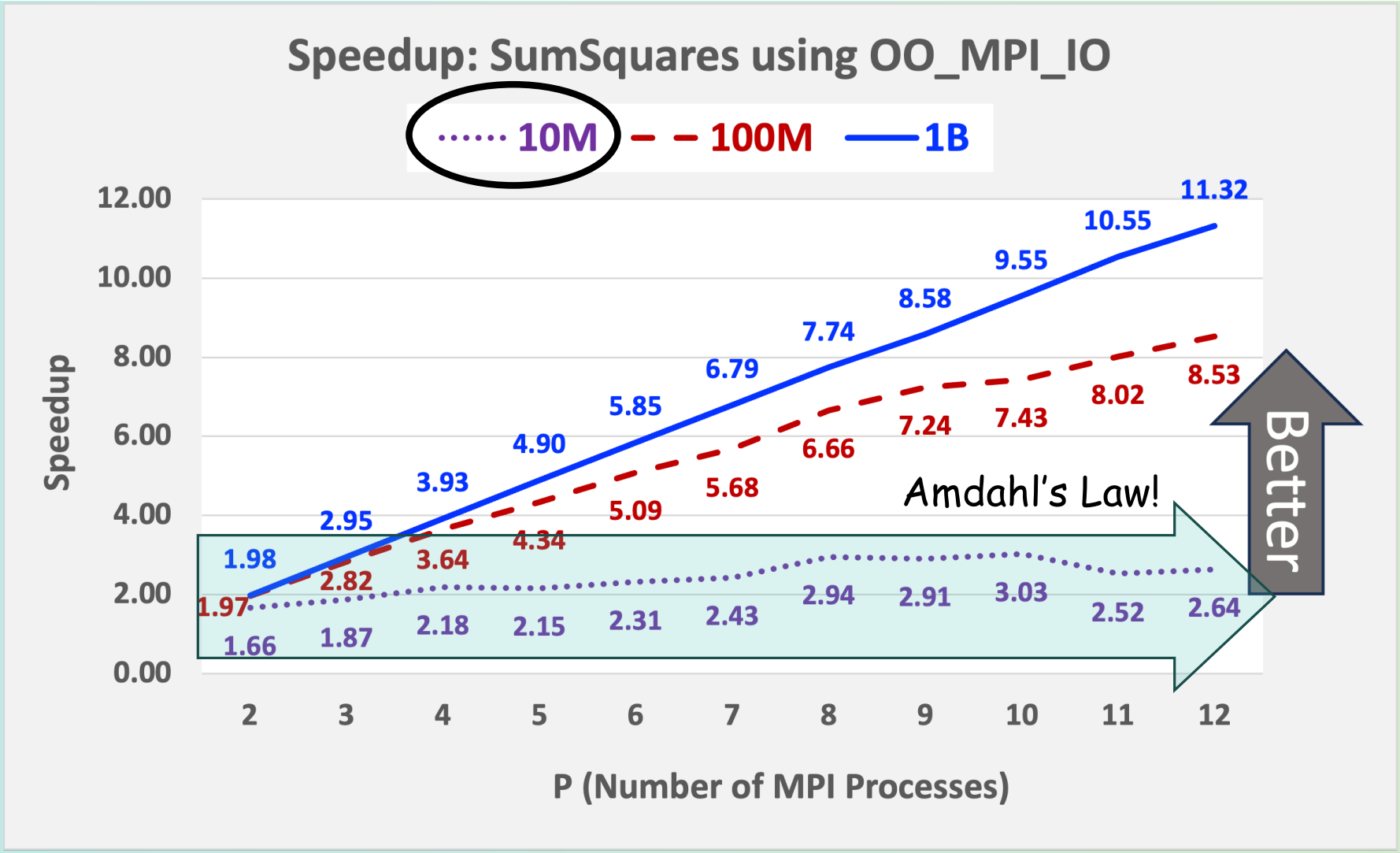
OO_MPI_IO

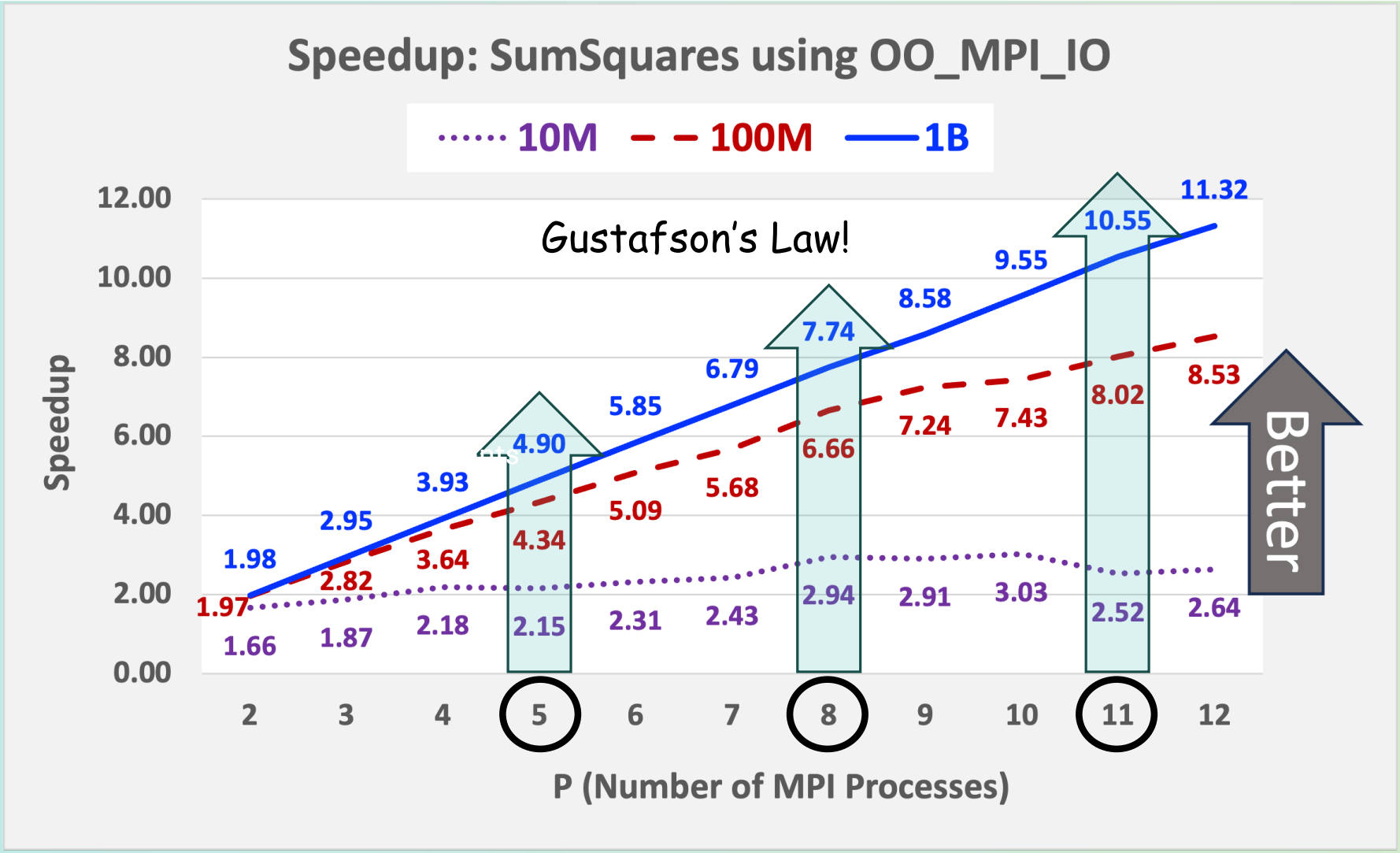
… is the name of the resulting package of abstractions:

With OO_MPI_IO available for the same sum-the-squares project, 77% of my 2023 HPC students (17/22) chose to use parallel I/O instead of read+scatter!

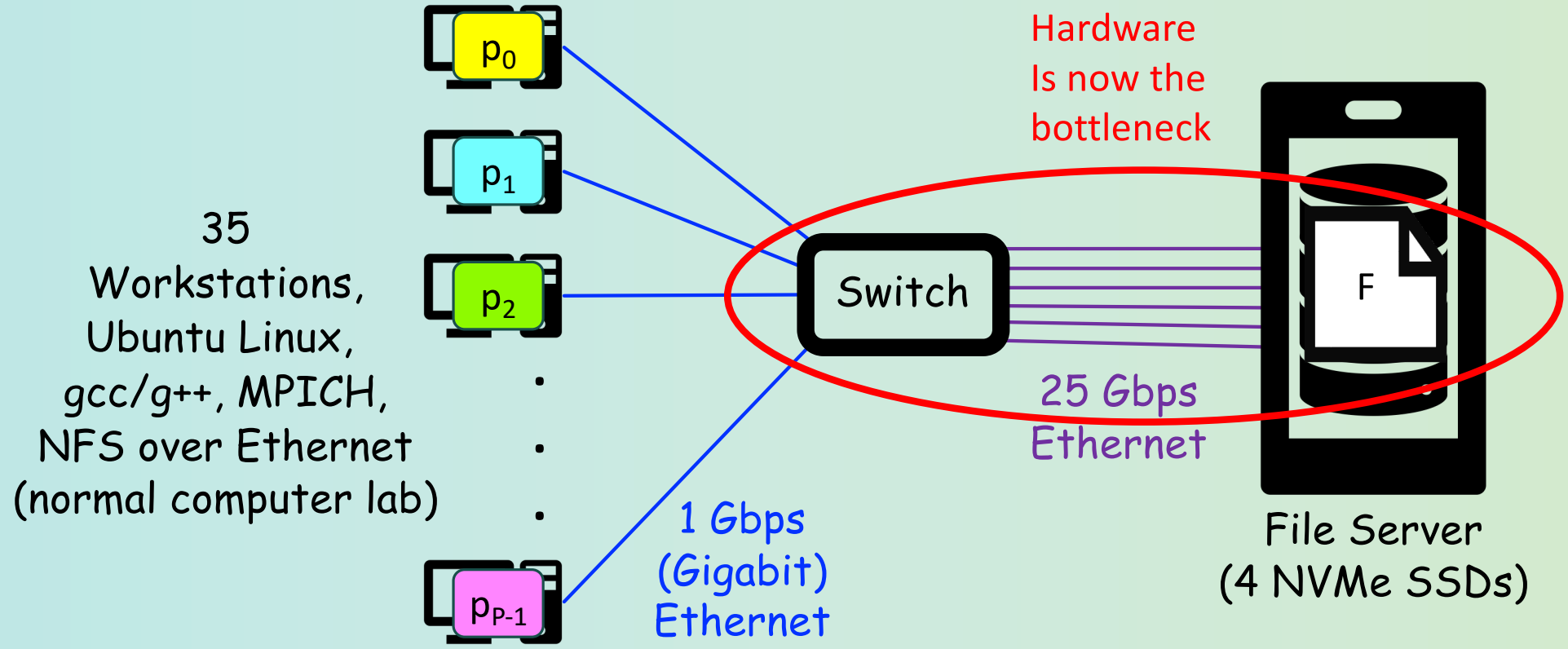



Time: SumSquares, 1B Doubles using OO_MPI_IO
Input-Step (P Parallel Reads)
SumSqrs
Time (secs)
8
7
6
5
4
3
2
1
0
7.18
3.63
2.44
1.83
1.47
1.23
1.06
0.93
0.84
0.75
0.68
0.63
1
2
3
4
5
6
7
8
9
10
11
12
P (Number of MPI Processes)
Better


Speedup: SumSquares using OO_MPI_IO
10M
100M
1B
Speedup
12.00
10.00
8.00
6.00
4.00
2.00
0.00
11.32
10.55
9.55
8.58
7.74
6.79
5.85
4.90
3.93
2.95
1.98
1.97
8.53
8.02
7.43
7.24
6.66
5.68
5.09
4.34
3.64
2.82
1.66
1.87
2.18
2.15
2.31
2.43
2.94
2.91
3.03
2.52
2.64
Amdahl's Law!
Better
2
3
4
5
6
7
8
9
10
11
12
P (Number of MPI Processes)


Speedup: SumSquares using OO_MPI_IO
10M
100M
1B
Gustafson's Law!
Speedup
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1.98
2.95
3.93
4.90
5.85
6.79
7.74
8.58
9.55
10.55
11.32
1.97
2.82
3.64
4.34
5.09
5.68
6.66
7.24
7.43
8.02
8.53
1.66
1.87
2.18
2.15
2.31
2.43
2.94
2.91
3.03
2.52
2.64
2
3
4
5
6
7
8
9
10
11
12
P (Number of MPI Processes)
Better

Parallel Hardware: Network of Workstations (NoW)



Conclusions

- When solving 'Big Data' problems, I/O can be a performance bottleneck
  → Reading numbers from a text file took 55X longer than using a binary file
- Sequential I/O significantly limits parallel speedup (Amdahl's Law)
  → Scattering in an MPI (distributed) environment exacerbates the problem
- Parallel I/O provides a way to bypass the sequential I/O bottleneck
  → MPI-IO is a *portable* parallel I/O mechanism
- Students can *actively experience Amdahl's and Gustafson's Laws* by using parallel I/O to solve a 'Big Data' problem with different-sized input files
- OO_MPI_IO provides *easy-to-use C++ abstractions for parallel I/O*
  → Students find it *less intimidating* than MPI-IO
  → Not limited to MPI; also permits parallel I/O in *multithreaded apps*

Finally…

- OO_MPI_IO is freely available via Github:
  - https://github.com/joeladams/OO_MPI_IO
  - Or contact me: adams@calvin.edu
- My thanks to:
  - NSF-DUE #1822486, whose support made this work possible
  - You, for your kind attention!