TECHNISCHE UNIVERSITÄT DARMSTADT

Large-Scale Parallel Computing

Aamer Shah

shah@cs.tu-darmstadt.de

EXERCISE 3

Exercise 3

- Four programs that perform matrix multiplication
  - Sequential program (base case)
  - Three parallel MPI versions

- Had to be compiled and executed on Lichtenberg-cluster
  - Measure the execution time
  - And time spent in other program sections

  - How many succeeded in running it?
  - How many can access the cluster?

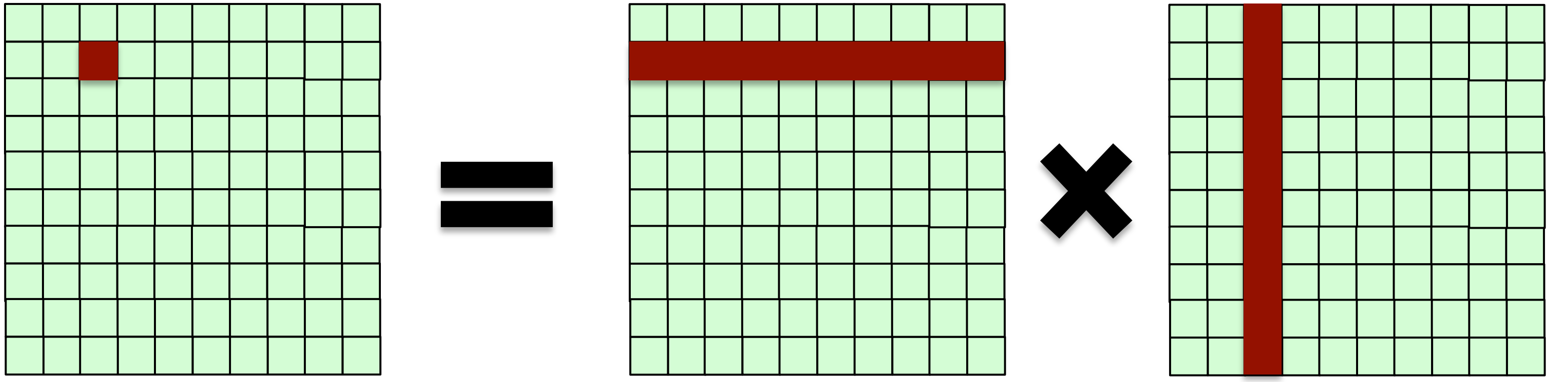
Matrix multiplication

- Matrix M1 size [M, N]
- Matrix M2 size [N, K]

M1 x M2

Columns of M1 must be equal to rows of M2

$$\text{Prod} = P_{(i,j)} : \Sigma\ (M1_{(i,t)} \times M2_{(t,j)}) \text{ where } 1 \le t \le N$$



Sequential program

Output: (time in nanoseconds)

Starting app at time: 1447778337607808256

Time spent in allocation: 333312

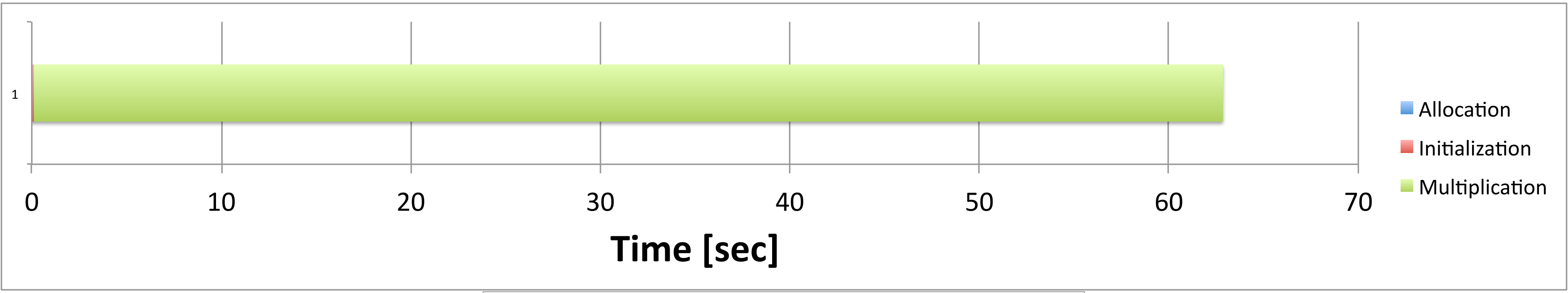
Time spent in initialization: 110592512

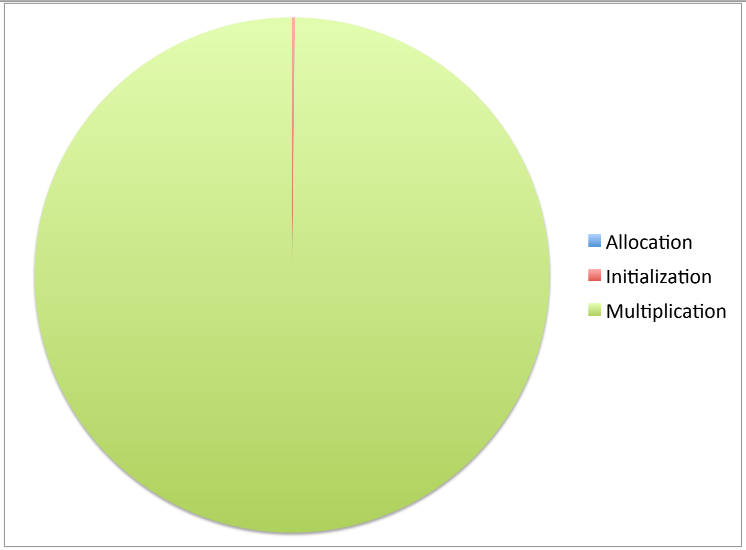
Time spent in multiplication: 74643288576

Total time spent in application: 74754214400

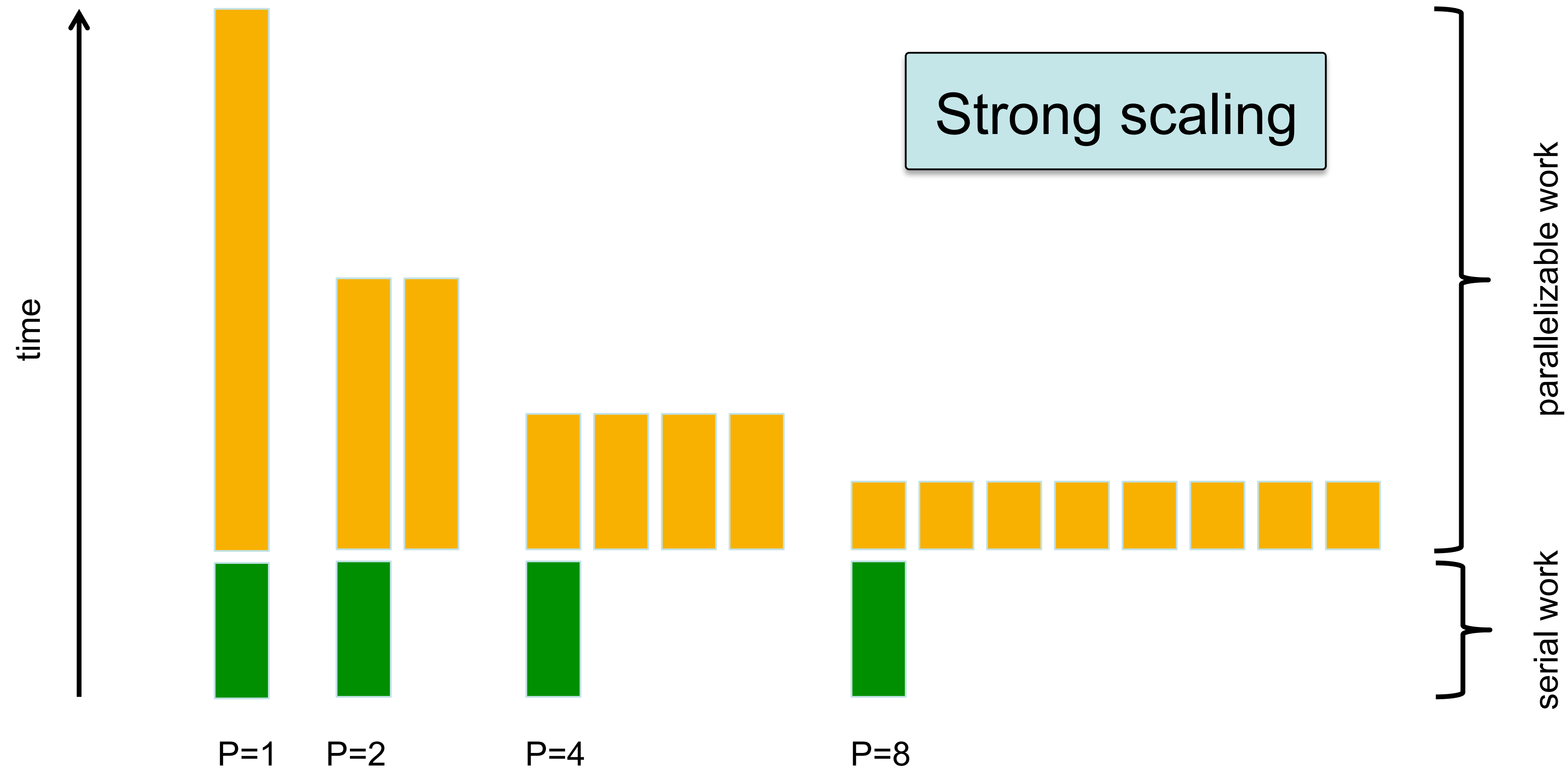
Execution time of sequential program

- Time distribution

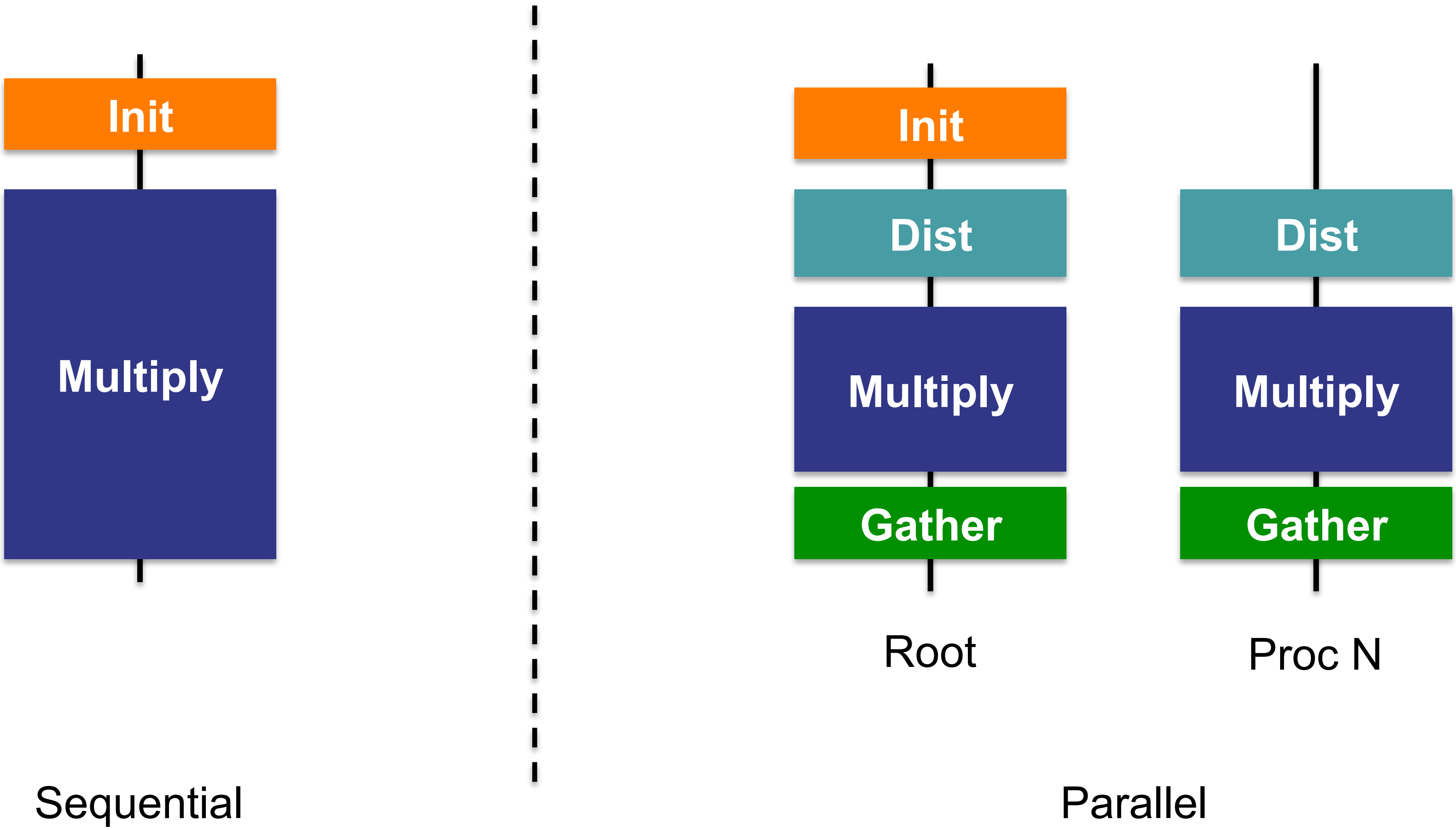




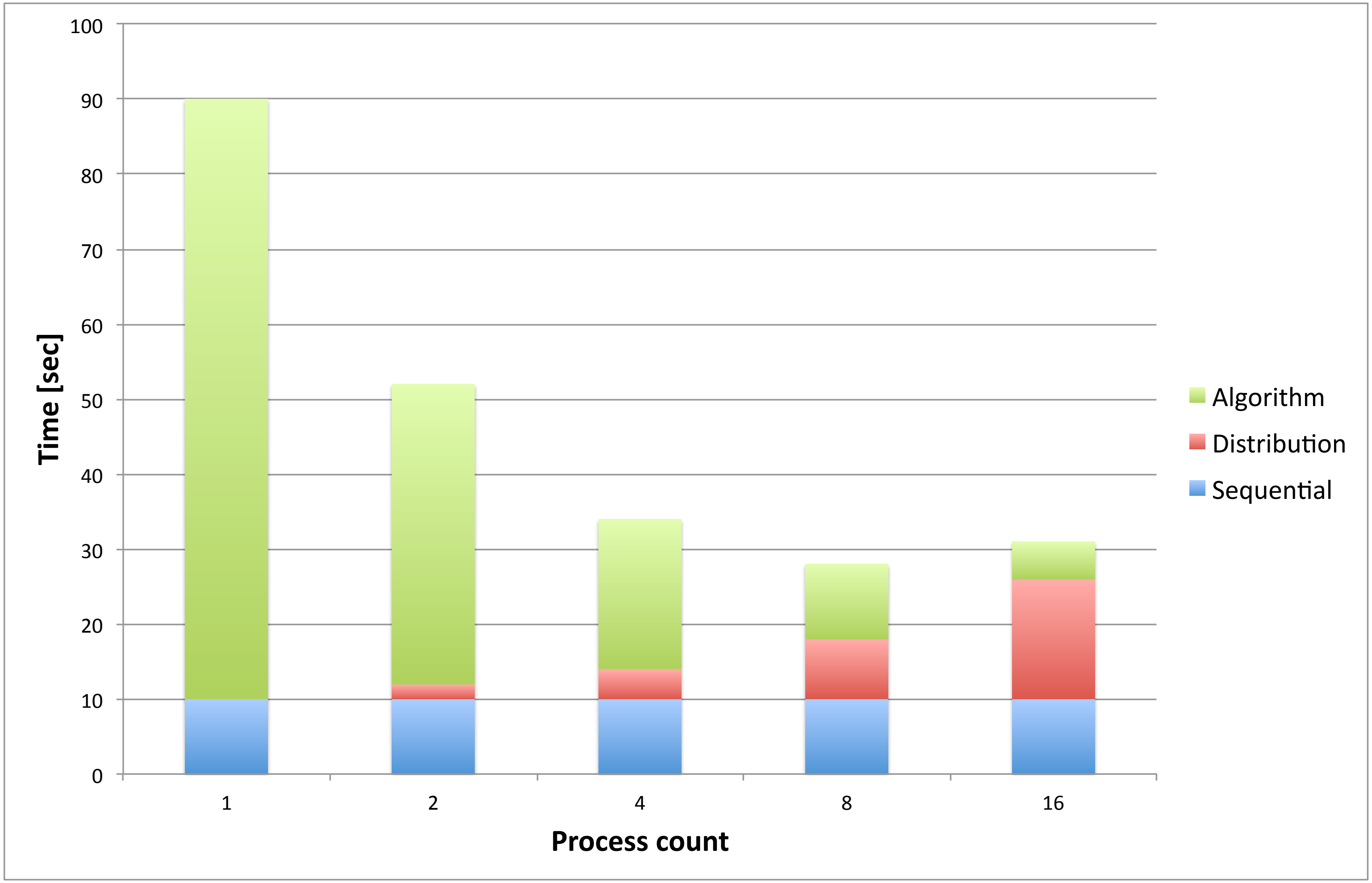
Amdahl's law (3)



Process tasks – sequential vs parallel

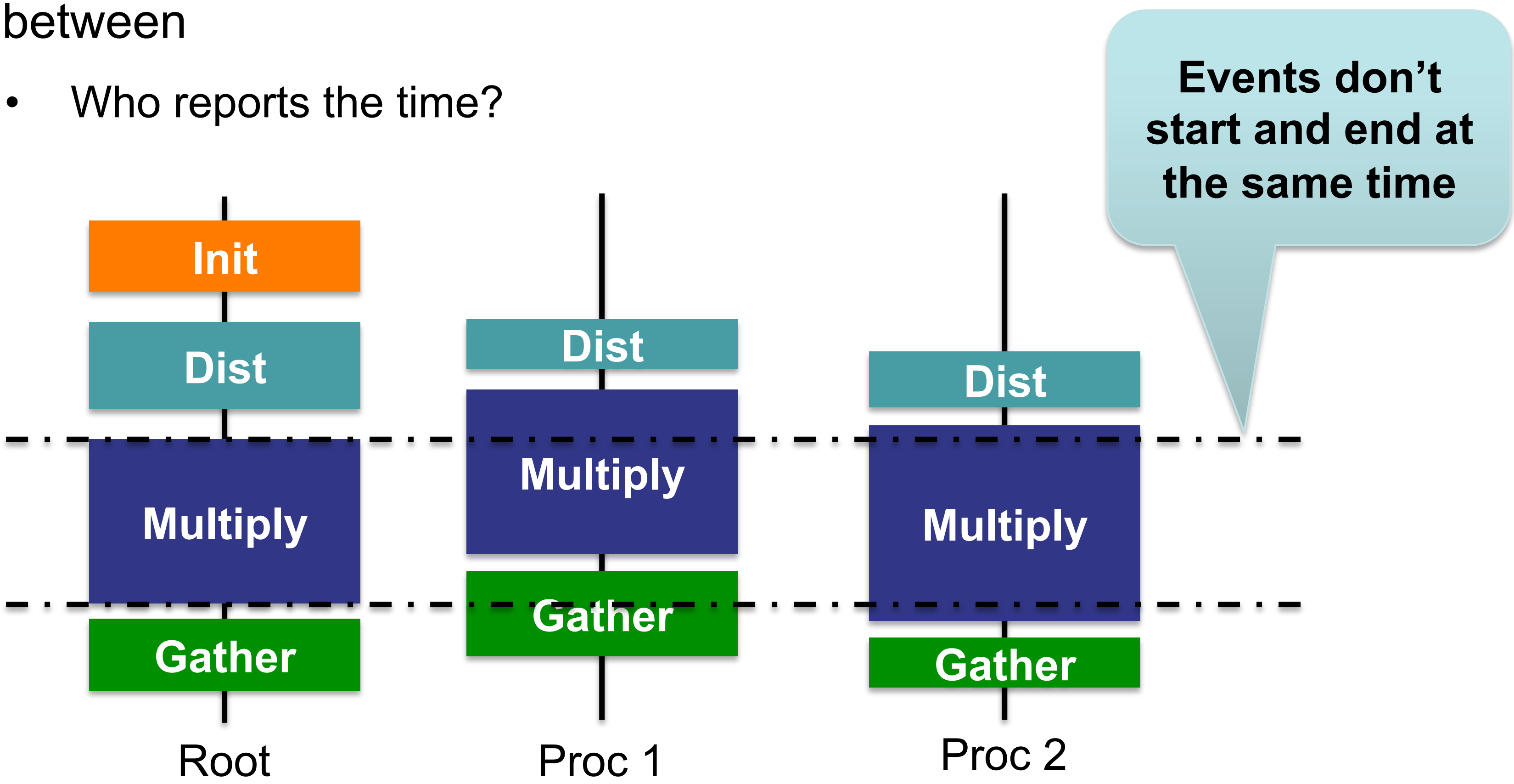


Application scaling



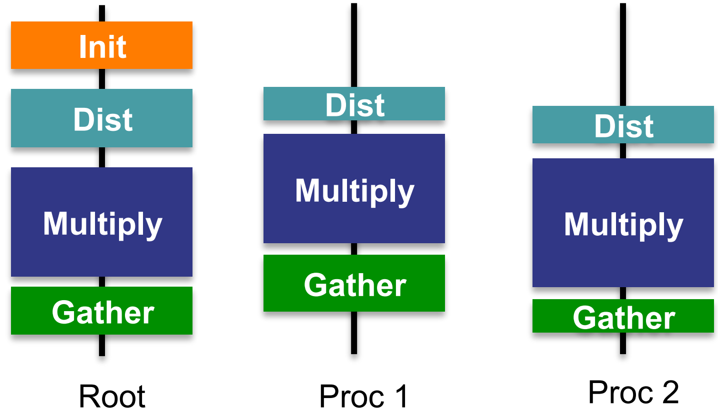
Reporting time

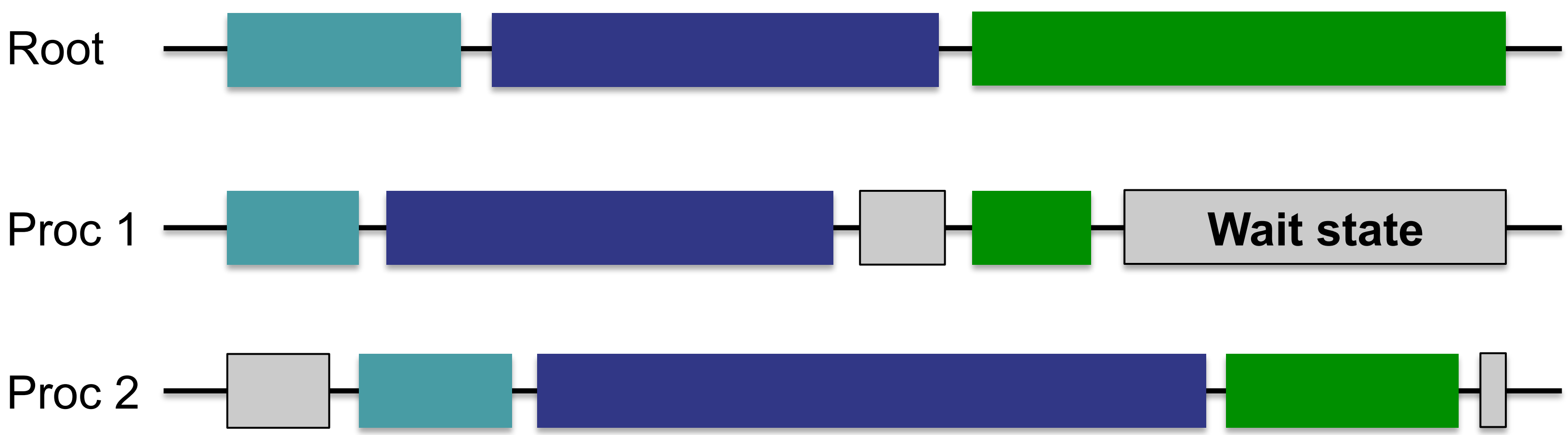
- Multiple independent processes, with synchronization in between
  - Who reports the time?



Reporting time

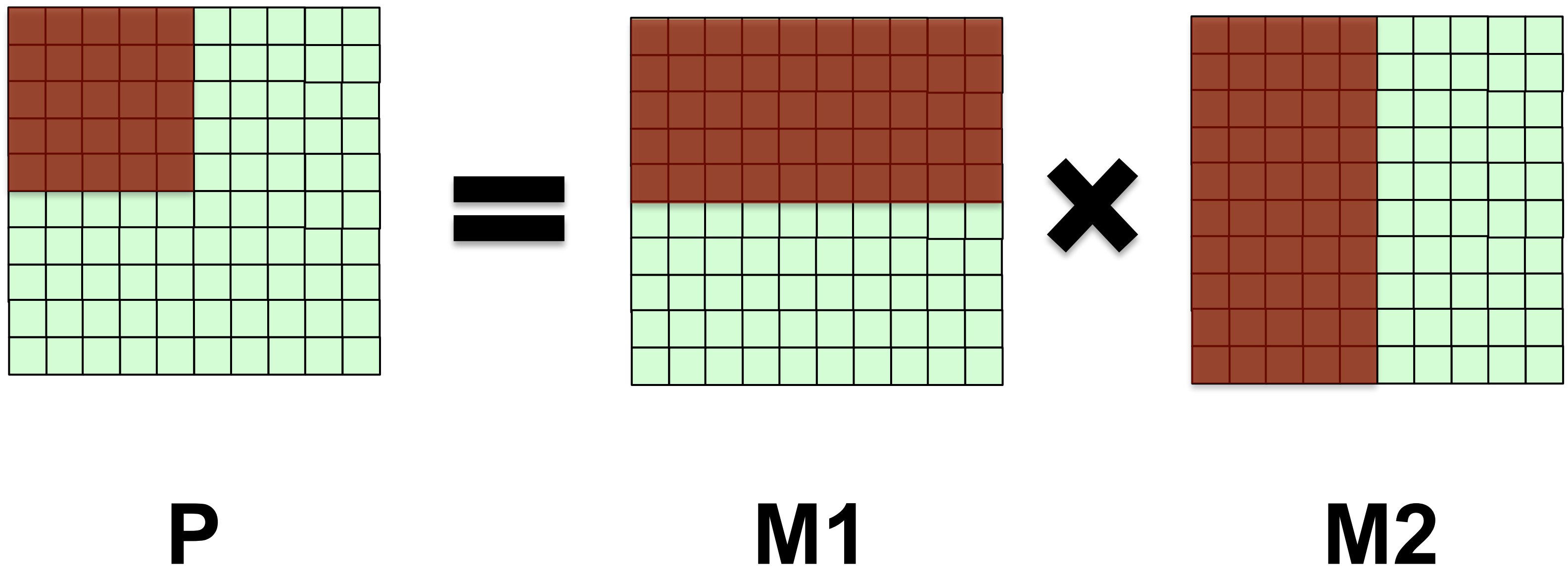
- Events order
  - Imbalance in computation can falsely be reported as time spent in data gathering





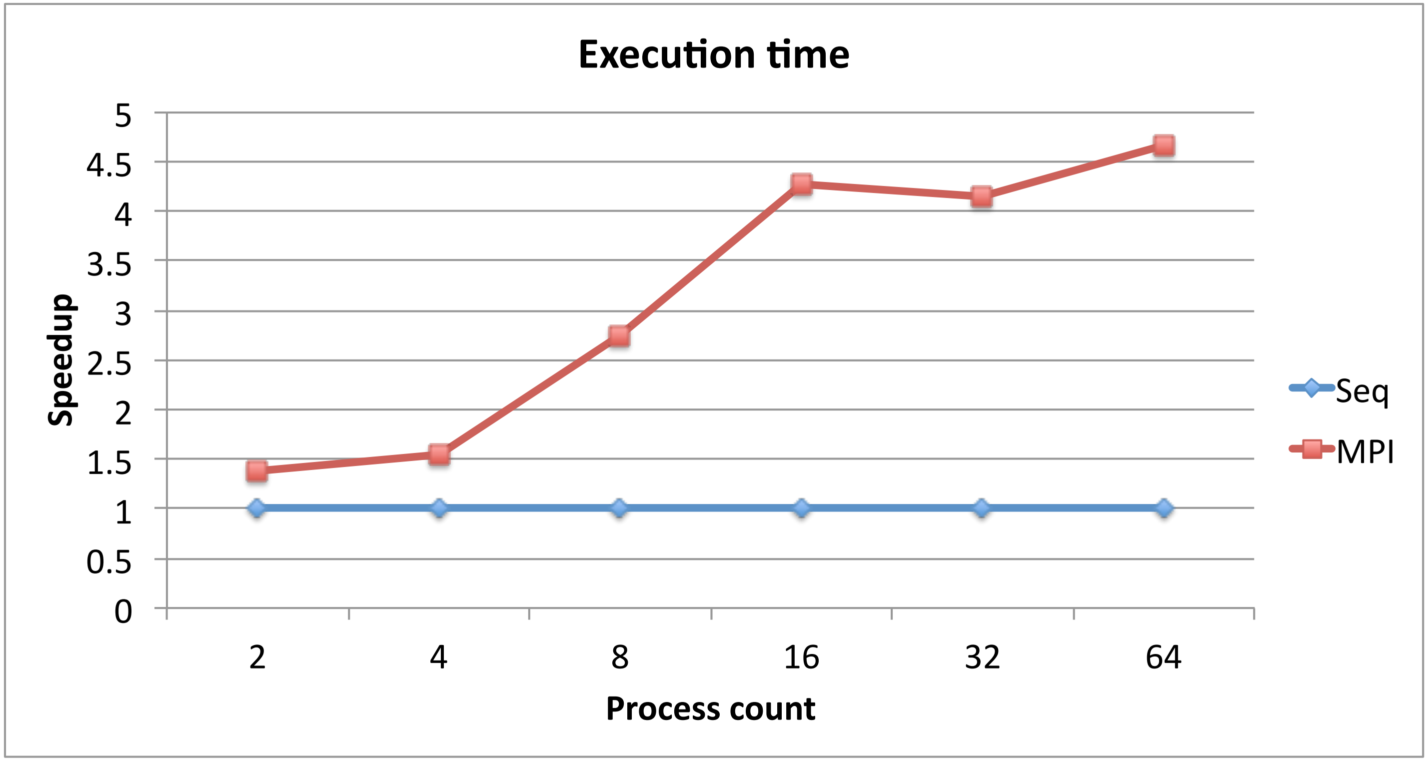
Data distribution

- Tile-wise distribution of data among processes

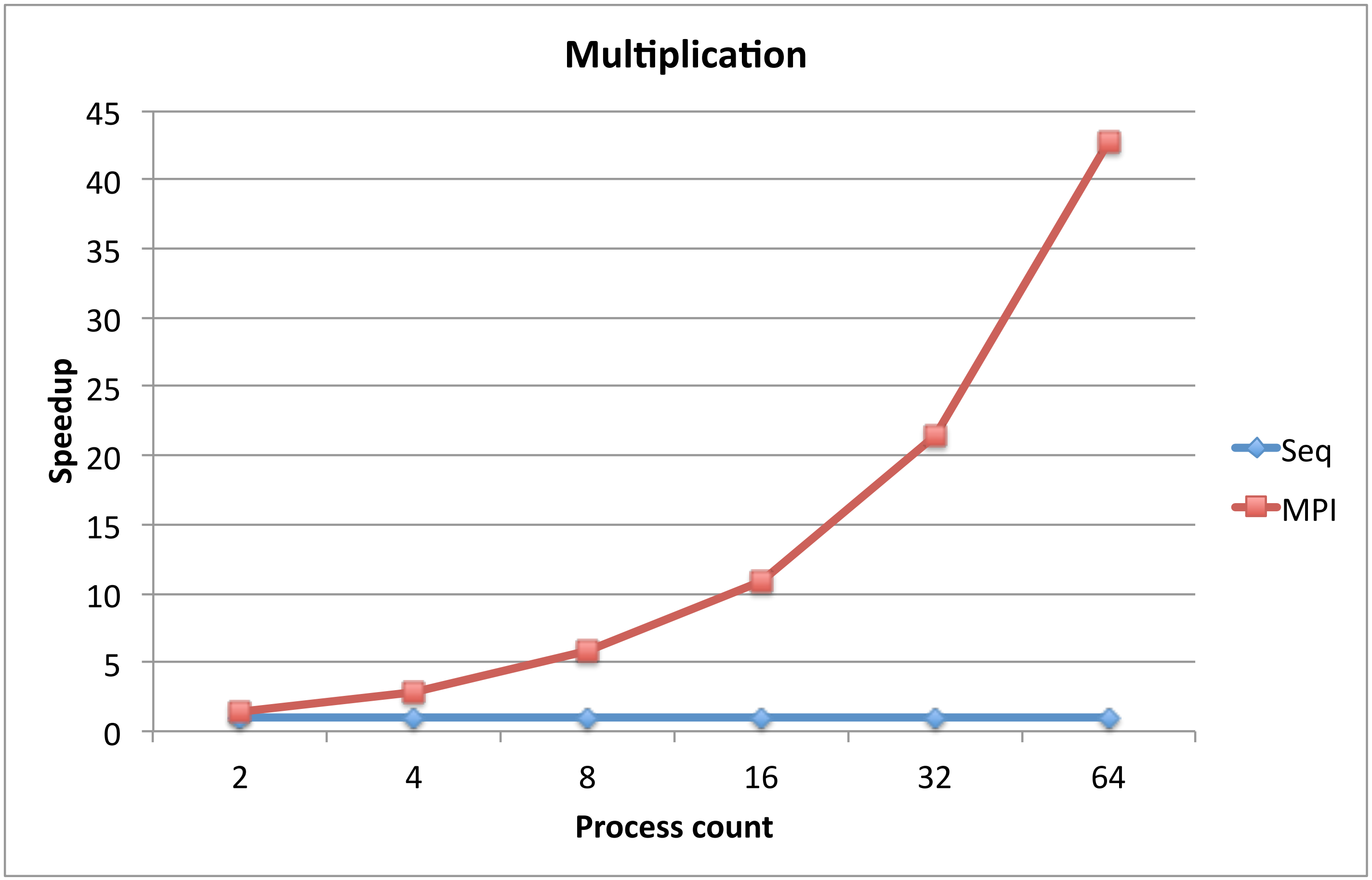


MPI - scaling

$$\text{Speedup} = \frac{\text{time parallel program}}{\text{time sequential program}}$$

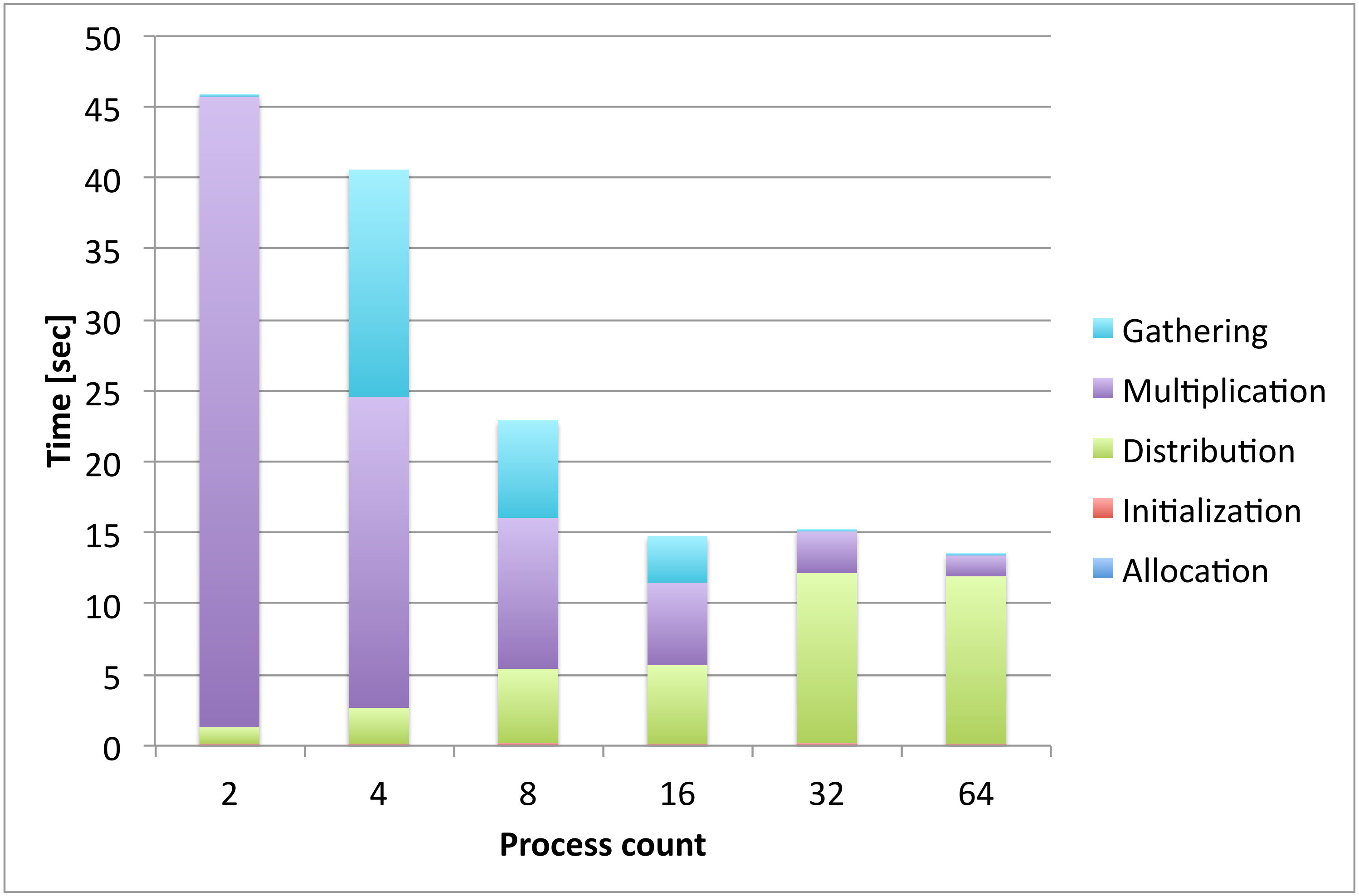


MPI - scaling



Multiplication
Speedup
45
40
35
30
25
20
15
10
5
0
2
4
8
16
32
64
Process count
Seq
MPI

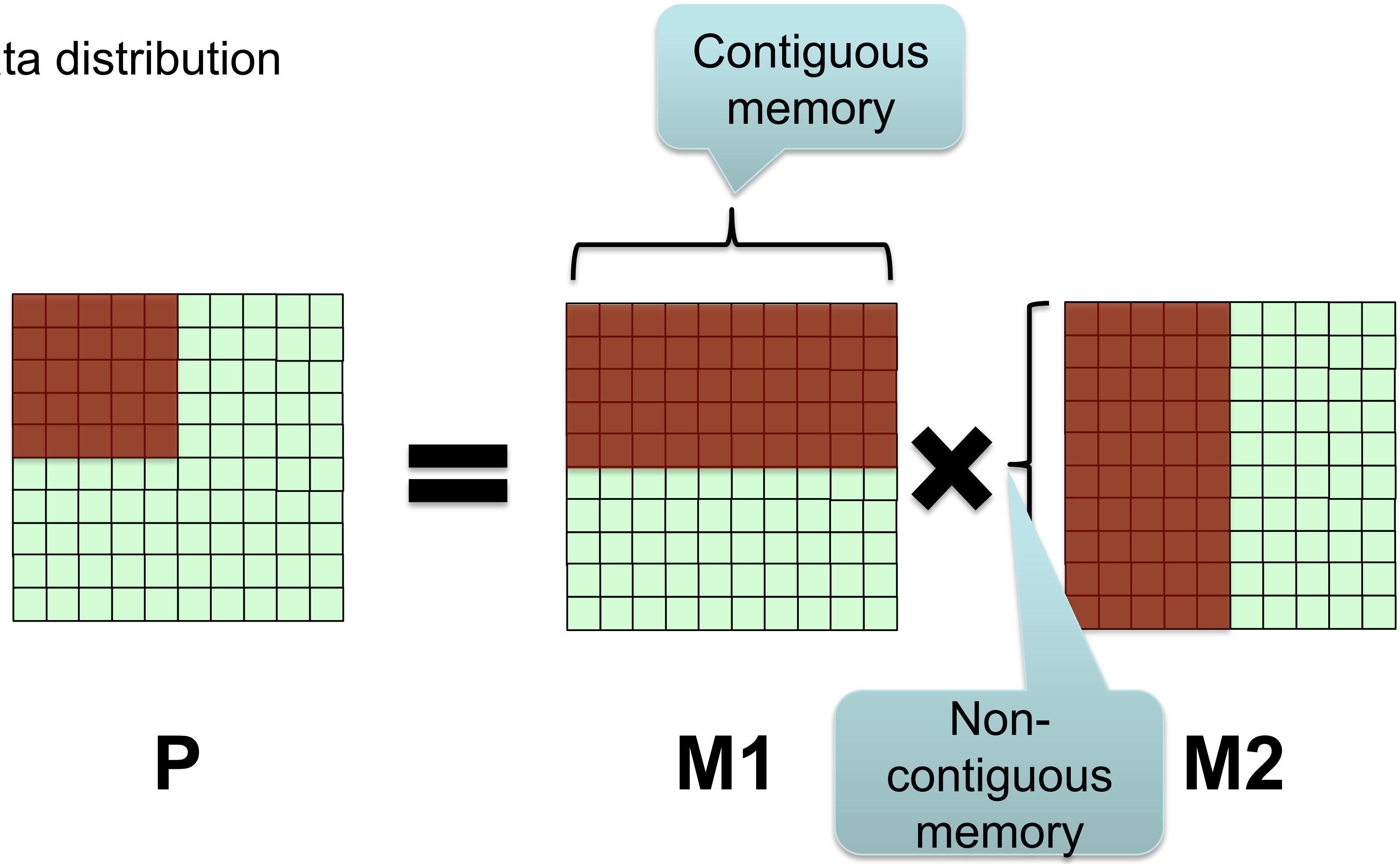
MPI – scalability graph



MPI – scalability problems

TECHNISCHE UNIVERSITÄT DARMSTADT

- Data distribution



MPI – scalability problem

TECHNISCHE UNIVERSITÄT DARMSTADT

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Columns

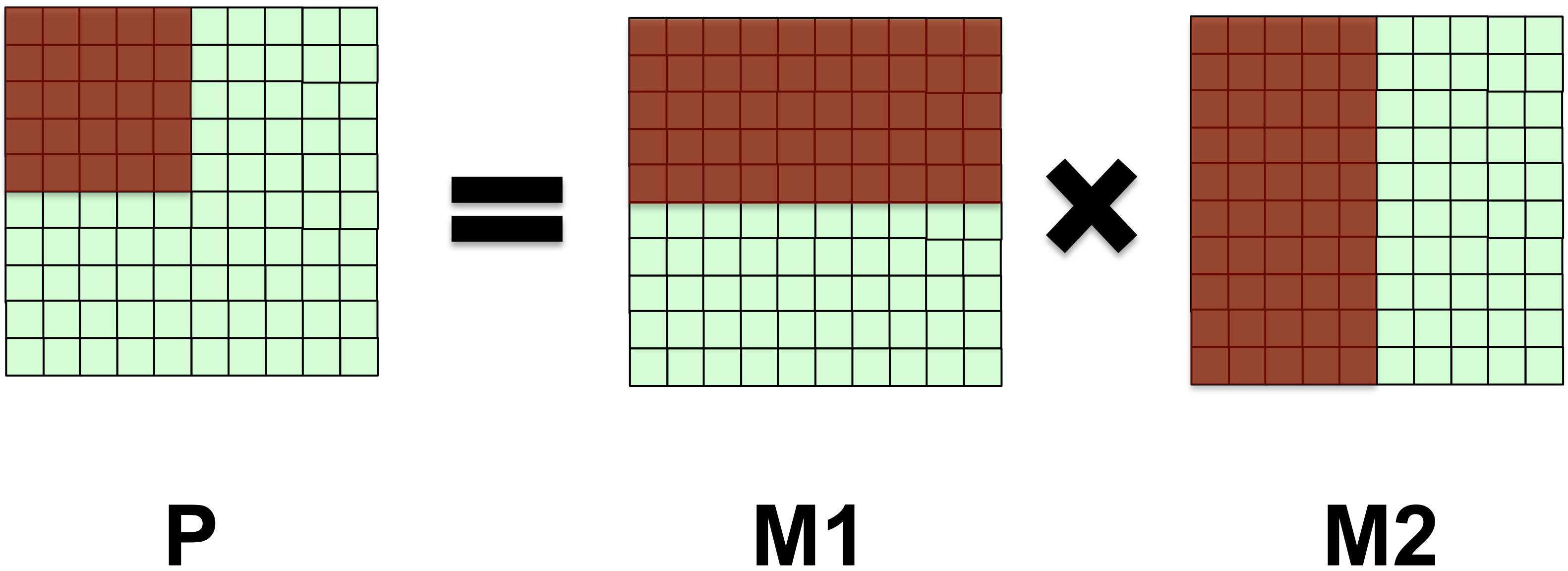
```
for(j = 0; j < cols_group_size; j++)
{
   int k;
   for(k = 0; k < DIM_LEN; k++)
   MPI_Send(&mat[k][col_disp * cols_group_size + j], 1, MPI_CHAR, i, 0, MPI_COMM_WORLD);
}
```

Rows

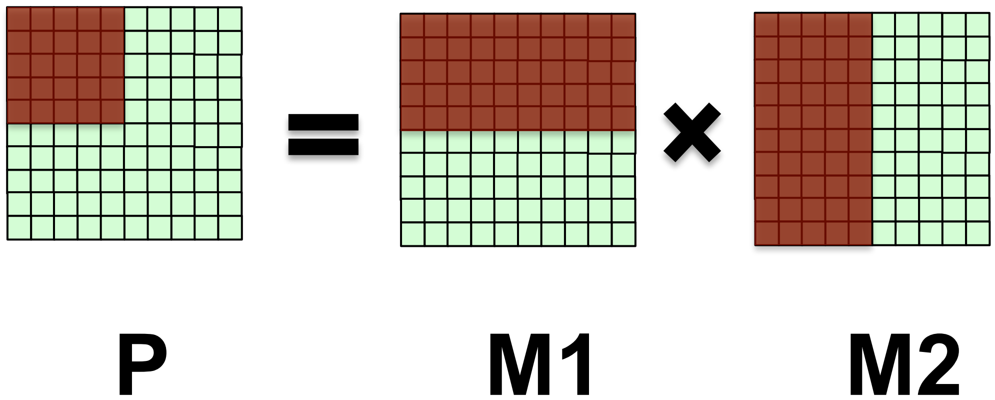
```
for(j = 0; j < row_group_size; j++)
{
   int row_ind = row_disp * row_group_size + j;
   MPI_Send(lmat[row_ind], DIM_LEN, MPI_CHAR, i, 0, MPI_COMM_WORLD);
}
```

MPI – scalability problem

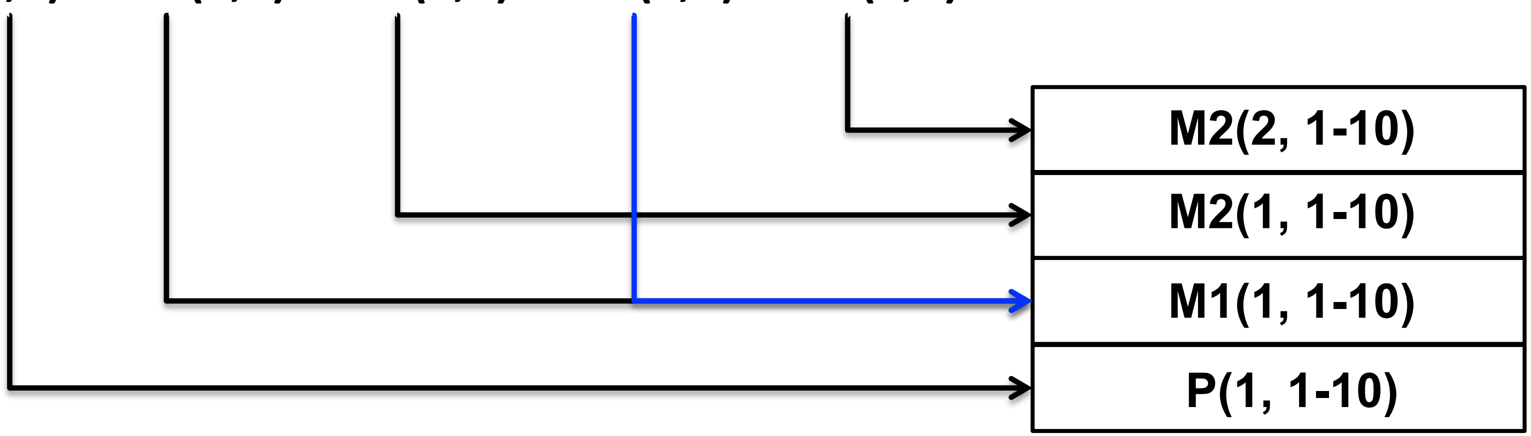
- Matrix multiplication
  - Non-contiguous memory access
  - Cache misses on M2



MPI – scalability problem



P(1,1) = M1(1,1) x M2(1,1) + M1(1,2) x M2(2,1) + …



MPI – scalability problem

Solution?

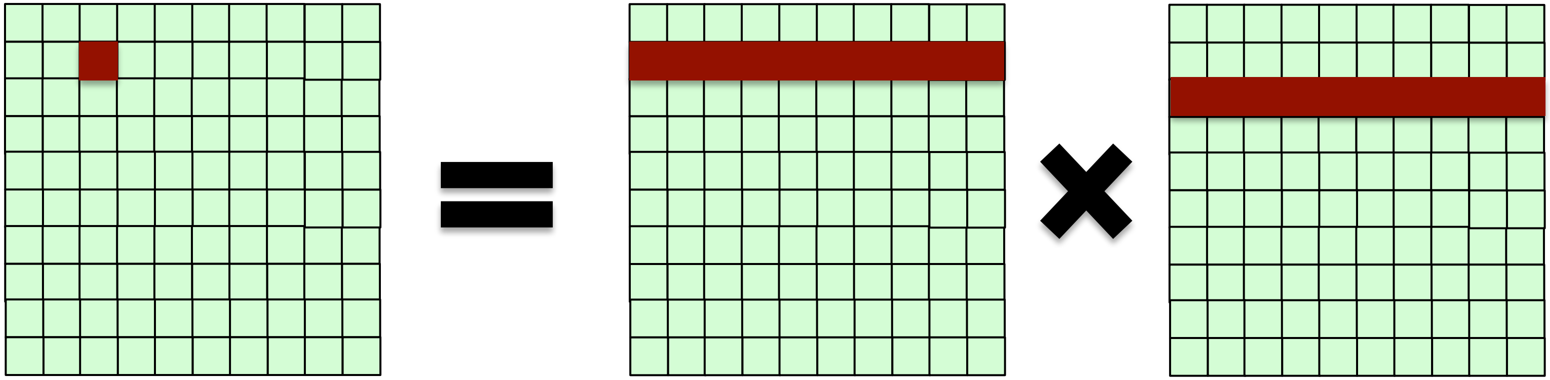
MPI – scalability problem - transpose

- Matrix M1 size [M, N]
- Matrix M2 size [N, K]   -> $M2^T$ size [K, N]

M1 x M2

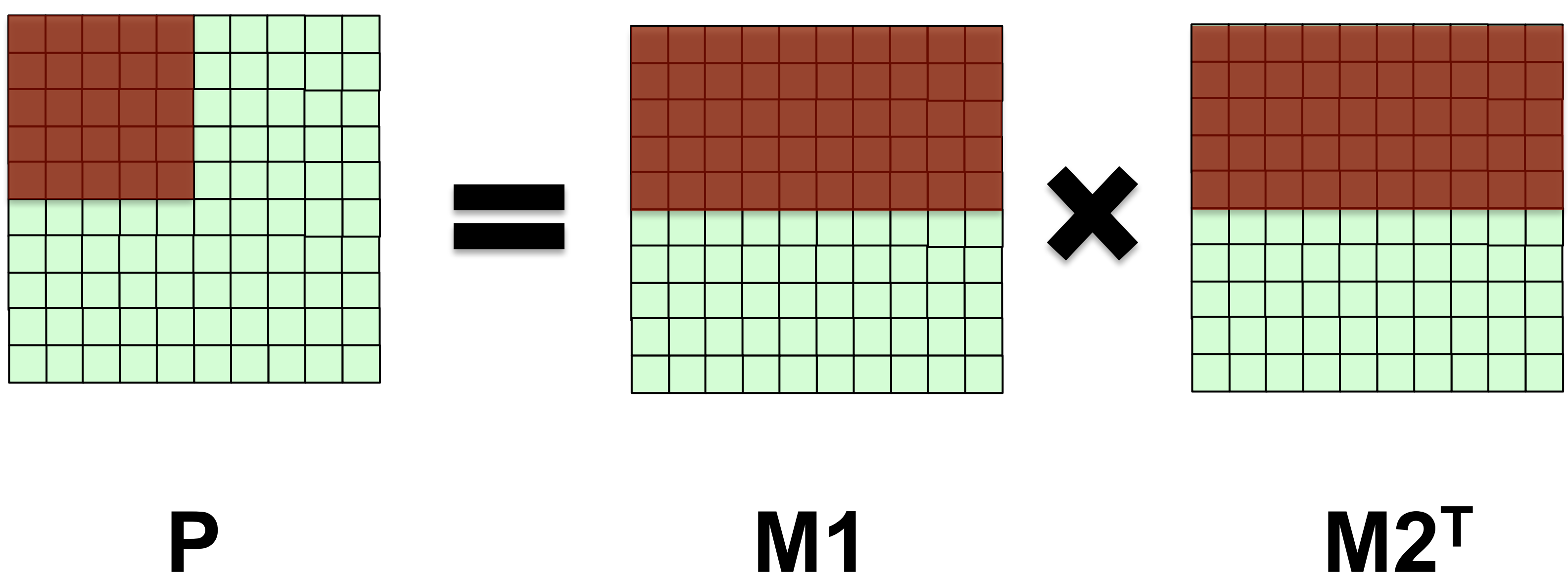
Columns of M1 must be equal to rows of M2

$$Prod = P_{(i,j)} : M1_{(i,t)} \times M2^T_{(j,t)} \text{ where } 1 \le t \le N$$



MPI – scalability problem

- Use transpose of M2
  - Solves data distribution problem
  - Solves cache misses problem



Trans – data distribution

TECHNISCHE UNIVERSITÄT DARMSTADT

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Columns

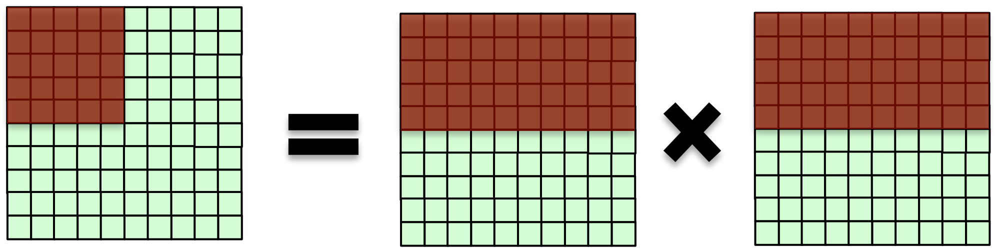
```
int col_disp = (i % col_div);
for(j = 0; j < cols_group_size; j++)
{
   MPI_Send(mat[col_disp * cols_group_size + j], DIM_LEN, MPI_CHAR, i, 0,
   MPI_COMM_WORLD);
}
```

Rows

```
for(j = 0; j < row_group_size; j++)
{
   int row_ind = row_disp * row_group_size + j;
   MPI_Send(lmat[row_ind], DIM_LEN, MPI_CHAR, i, 0, MPI_COMM_WORLD);
}
```

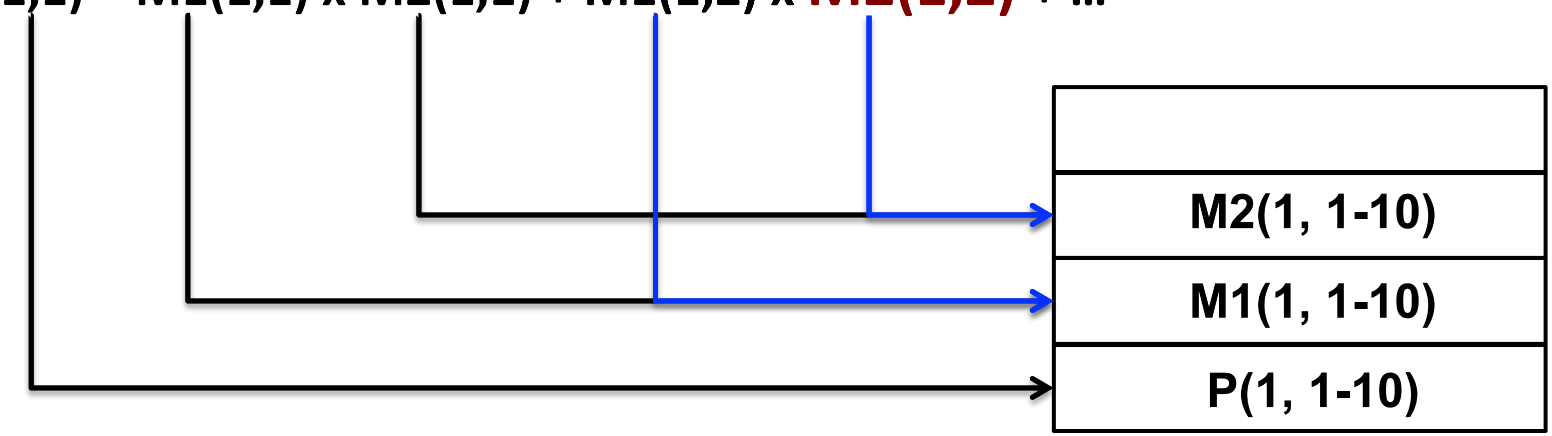
Trans – cache misses

TECHNISCHE UNIVERSITÄT DARMSTADT

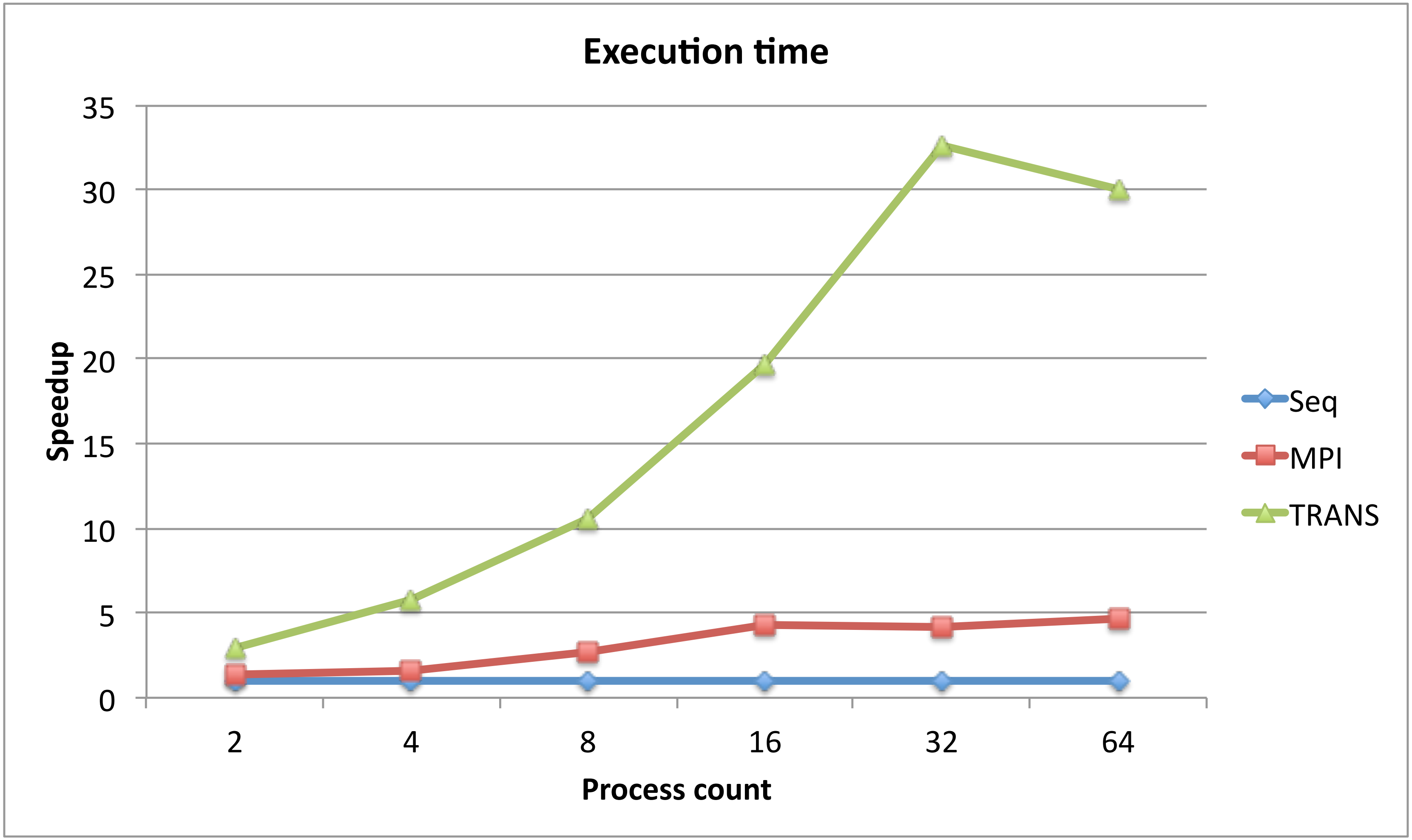


P **M1** **M2**

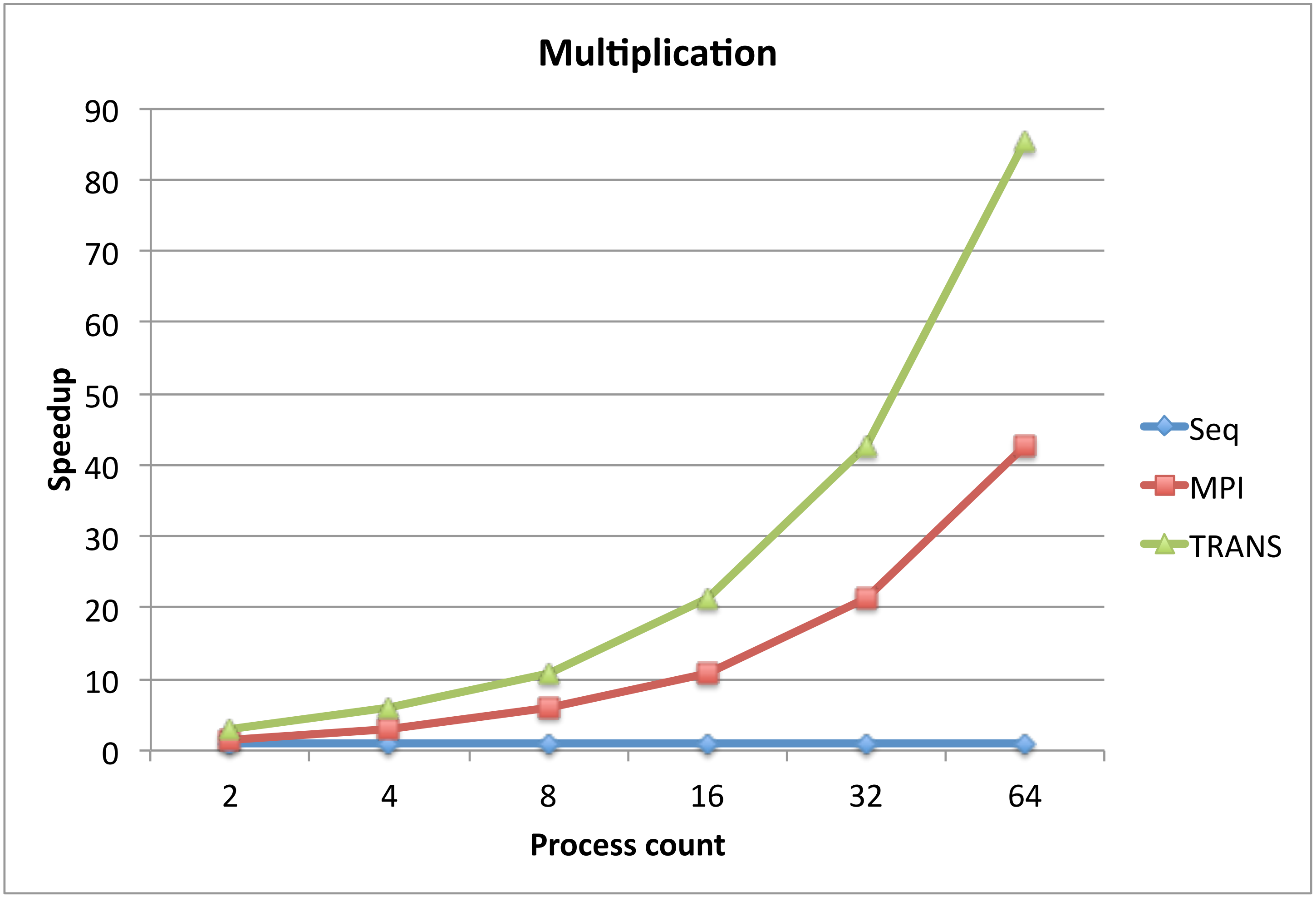
P(1,1) = M1(1,1) x M2(1,1) + M1(1,2) x M2(1,2) + …



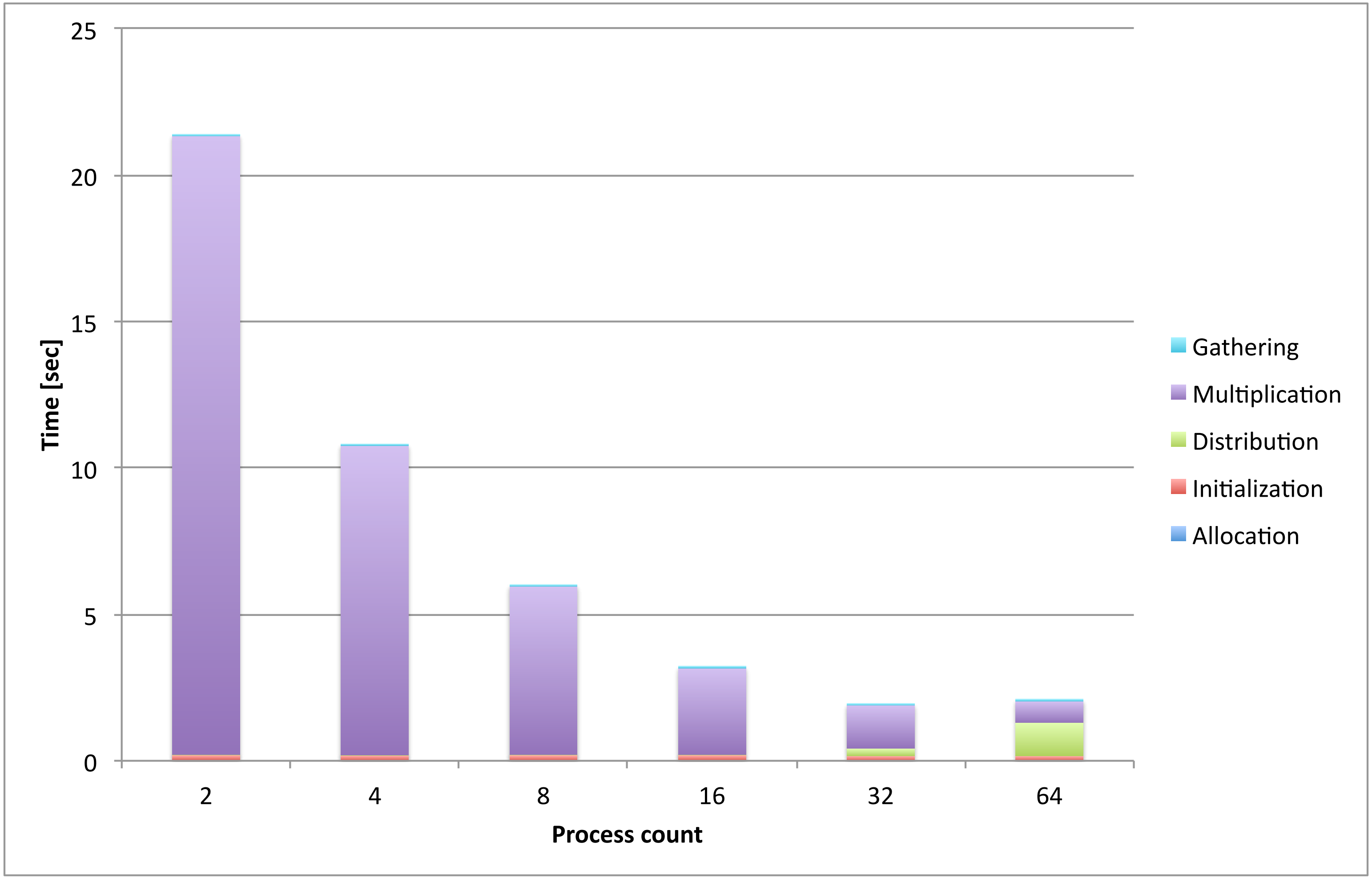
Trans - scaling



Trans - scaling



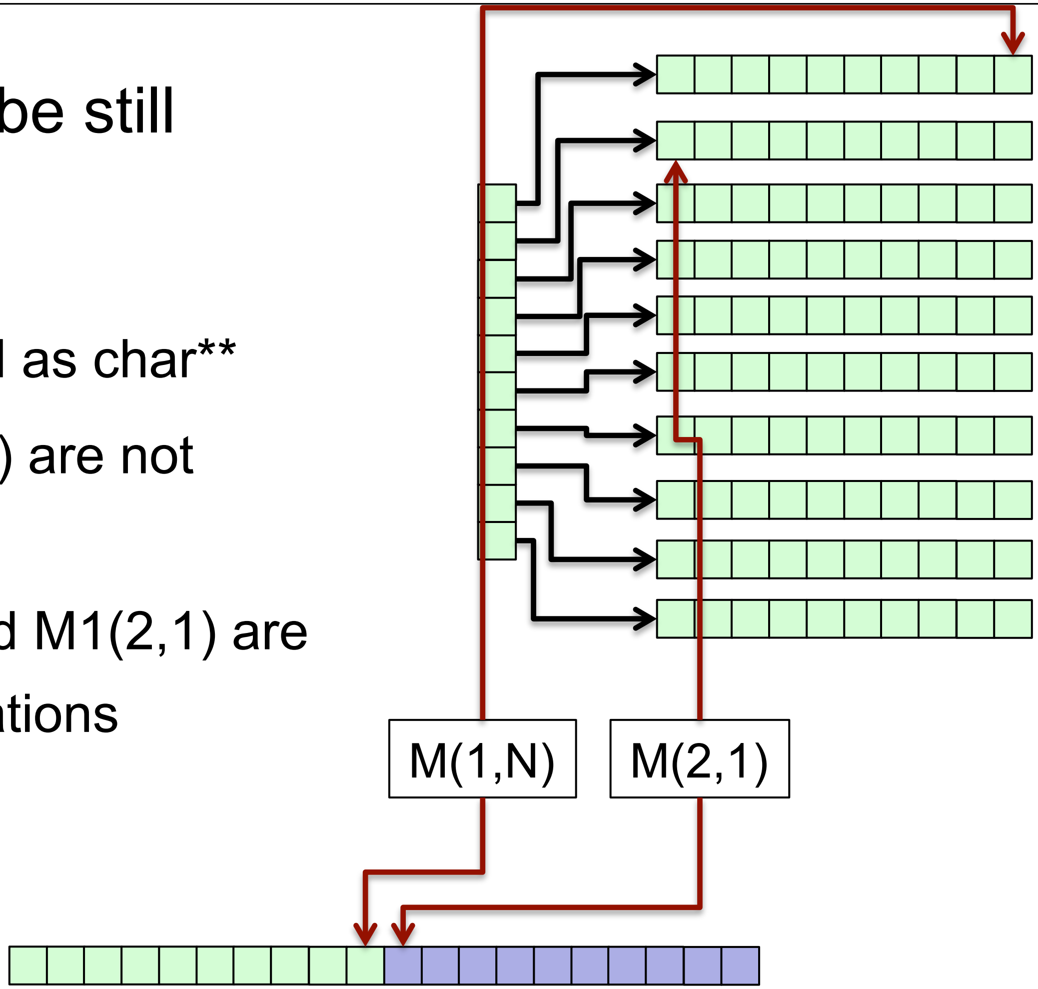
Trans - scalability



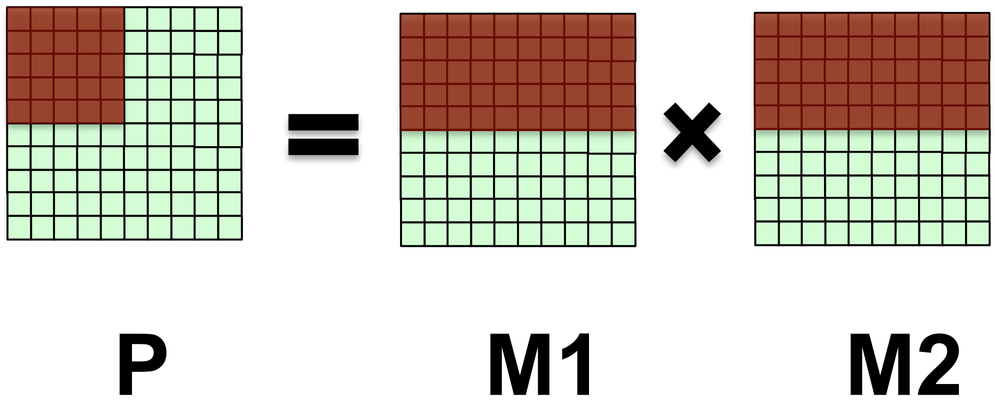
Any further improvements?

- Can the application be still improved?

- M1 and M2 are declared as char**
    - M1(1, N) and M1(2,1) are not continuous
- In 2D array, M1(1,N) and M1(2,1) are contiguous memory locations



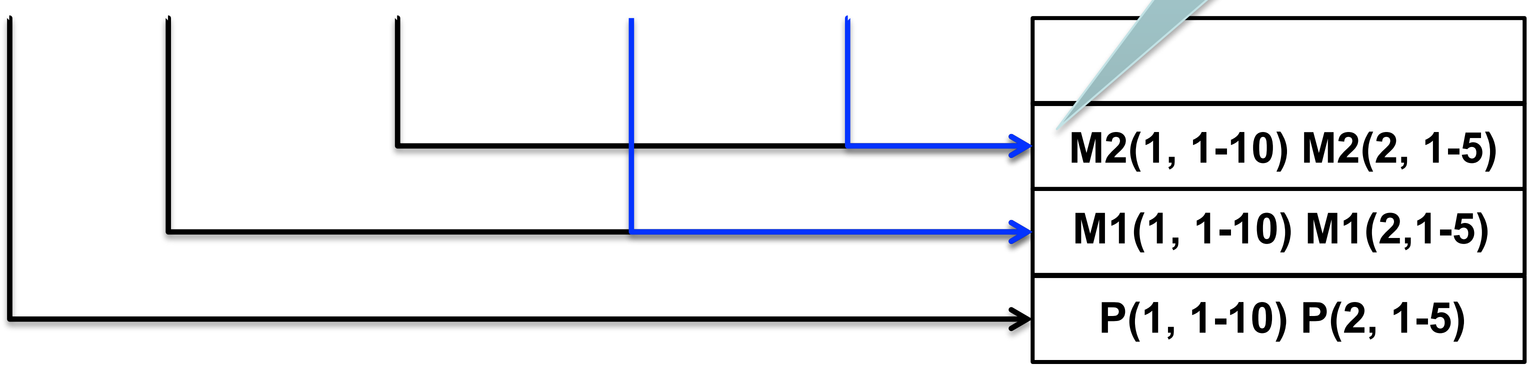
Potential optimization



- Cache miss if non-contiguous memory
- Cache hit if contiguous memory

P(1,1) = M1(1,1) x M2(1,1) + M1(1,2) x M2(2,1) + …

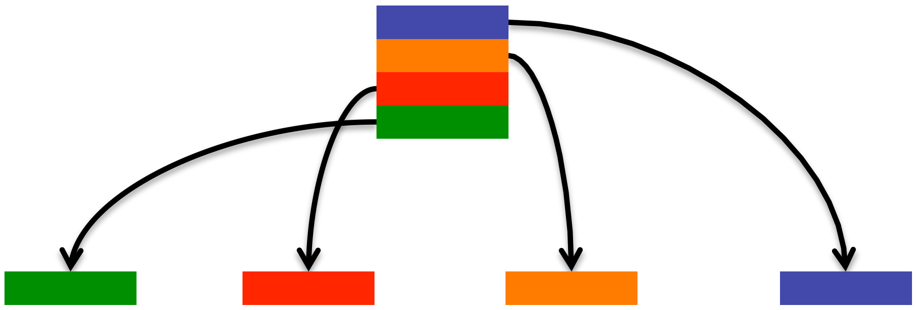
P(2,1) = M1(2,1) x M2(2,1) + M1(2,2) x M2(2,2) + …



Potential optimizations

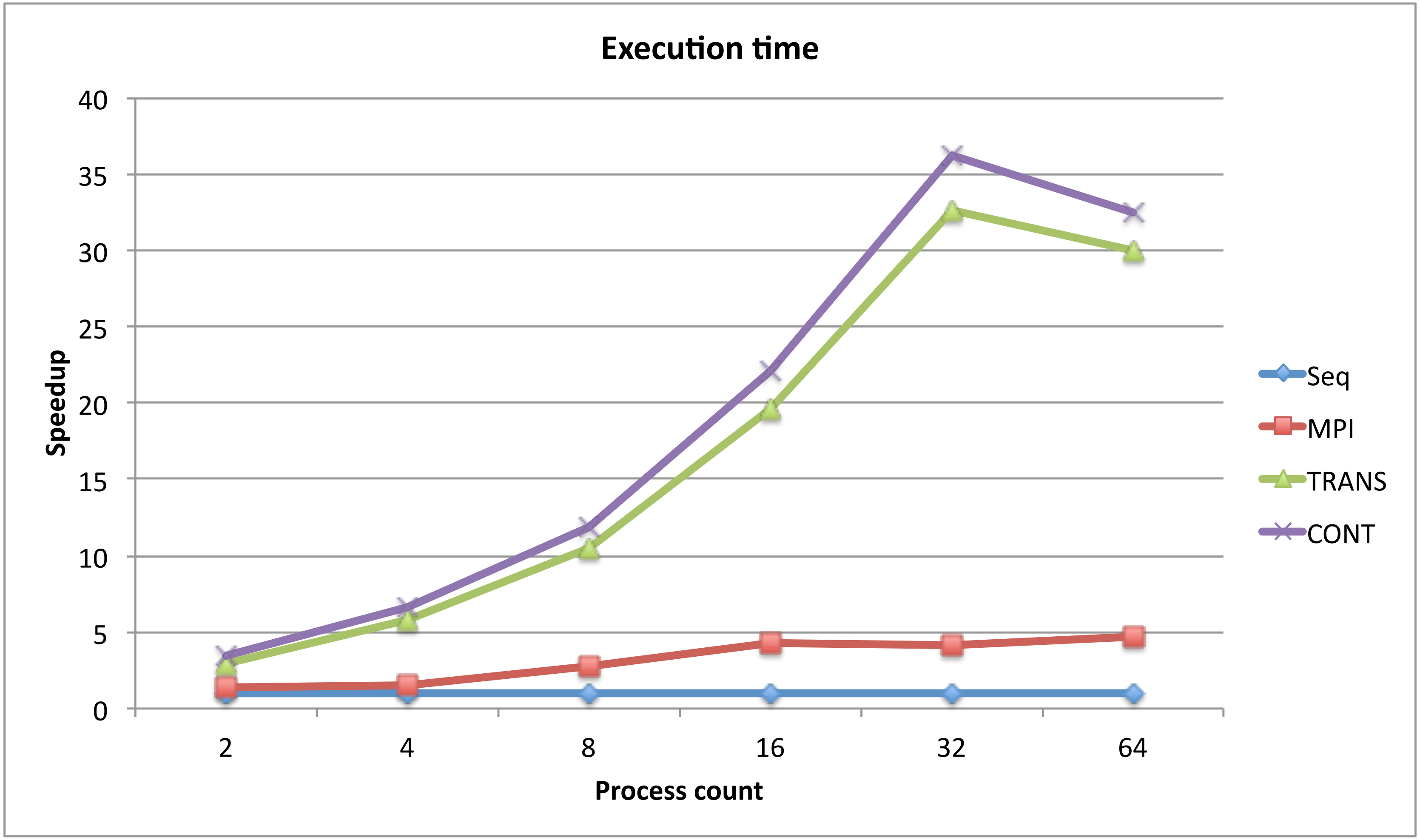
- For data distribution, MPI provides utility functions that have better performance

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

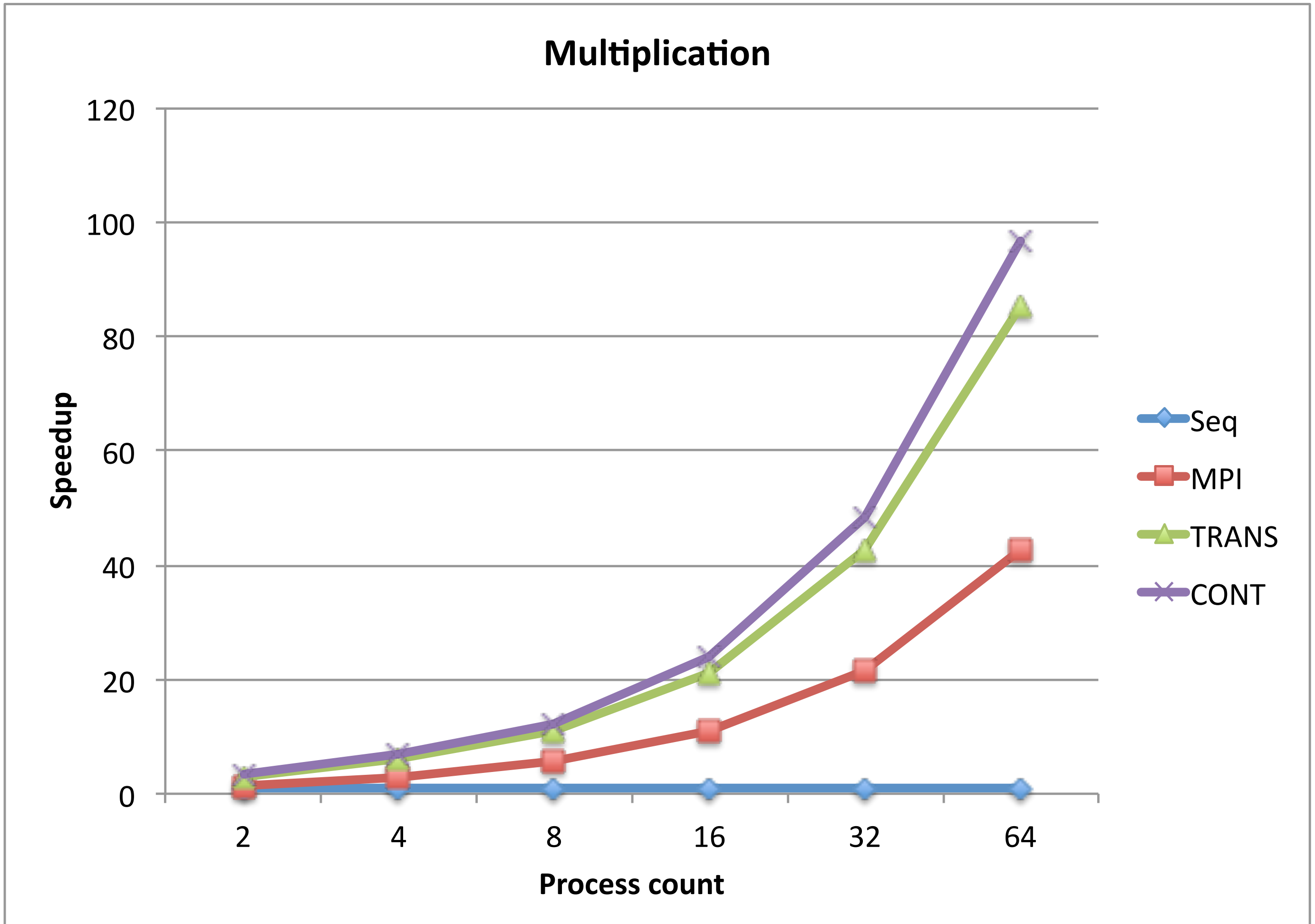


MPI_Scatterv(mat, scount, displacement, MPI_CHAR, mat, cols_group_size, MPI_CHAR, 0, MPI_COMM_WORLD);

Cont - scaling



Cont - scaling



Cont - scalability