PARALLEL NUMERICAL INTEGRATION USING ROMBERG'S METHOD

A comparison of two different techniques

By Gabriele Volani and Sanasar Hambardzumyan

January 2026

THE PROBLEM

- Approximation of a double integral over a triangular region using a two step Romberg-like techinque.
- Step 1: evaluate the function on an increasing number of points.
- Step 2: refine estimations using a Richardson Extrapolation.

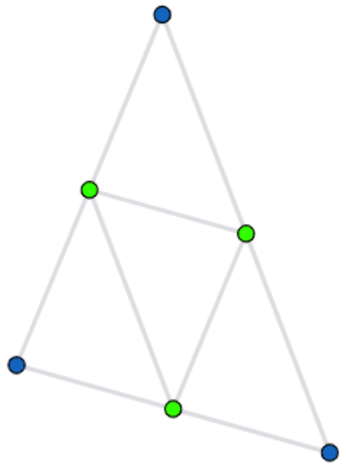
$$R[n][0] = \frac{A}{3 \cdot 4^n}\left(\sum_{(x,y)\in C} f(x,y) + \sum_{(x,y)\in E_n} f(x,y) + \sum_{(x,y)\in I_n} f(x,y)\right)$$

$$R[n][m] = \frac{2^m R[n][m-1] - R[n-1][m-1]}{2^m - 1}$$

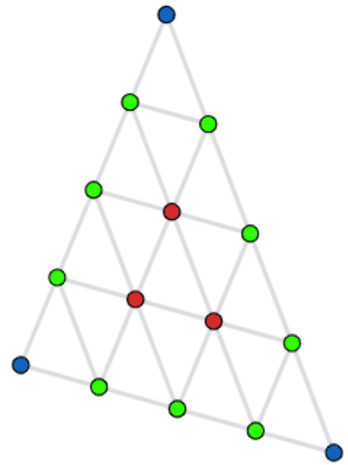
THE PROBLEM

n = 0

n = 1



n = 2



PARALLEL DESIGN

- Performances bottlekneck is in step 1…

TABLE I
SIZES OF E_n AND I_n

n	sub triangles	edge points (excl. vertices)	interior points
0	1	0	0
1	4	3	0
2	16	9	3
3	64	21	21
...	...	...	...
7	16384	381	8001
n	4^n	$3 \cdot 2^n - 3$	$\frac{(2^n+1)(2^n+2)}{2} - 3 \cdot 2^n$

- …functions evaluations are thousands.

ALGORITHM #1

- **Hybrid parallelization: MPI + OpenMP.**
- Master process computes at every step the set of edge nodes and interior nodes.
- Then it assigns a subset of nodes to all the processes.
- It distributes it (*MPI_Scatterv*).
- Other processes evaluate the function on the nodes assigned and compute local sums.
- *MPI_Reduce* to sum all local contributions to the master.
- Master process computes the Romberg's table.

ALGORITHM #1

- OpenMP parallelization:
  - generation of edge points by the master
  - generation of interior points by the master
  - evaluations of the function on the points assigned to a process

DATA DEPENDENCY

EDGE POINTS GENERATION

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
u	7	i	write	7	$i+1$	write	yes	OUTPUT	NO (put it as private)
pAB–pBC–pCA	11–14–17	i	write	11–14–17	$i+1$	write	yes	OUTPUT	NO (put them as private)
pAB–pBC–pCA	8–11–14	i	write	12–15–18	i	read	no	FLOW	NO (within same thread)
ec	18	i	write	12	$i+1$	read	yes	FLOW	**YES** (depends on previous iteration)

- The *ec* variable is the most critical: it behaves as a counter and it is accessed contemporary by multiple threads.
- Solution: precompute its values.

```
1  int t, ec=0;
2  double u;
3
4  Point pAB, pBC, pCA;
5  for (t = 1; t <= nm - 1; ++t) {
6      // Contribution of each point
7      u = (double) t / nm;
8
9      // Each point on the edge
10     // is a linear interpolation of its extremes
11     pAB = lerp(A, B, u);
12     edge[ec++] = pAB;
13
14     pBC = lerp(B, C, u);
15     edge[ec++] = pBC;
16
17     pCA = lerp(C, A, u);
18     edge[ec++] = pCA;
19 }
```

DATA DEPENDENCY

INTERIOR POINTS GENERATION

Memory Location	Earlier Statement Line	Iter.	Access	Later Statement Line	Iter.	Access	Loop-carried?	Kind of dataflow	Issue
u	10	i	write	10	$i+1$	write	yes	OUTPUT	NO (put it as private)
v	11	i	write	11	$i+1$	write	yes	OUTPUT	NO (put it as private)
ib	8	ia	write	8	$ia+1$	write	yes	OUTPUT	NO (put it as private)
p	13	i	write	13	$i+1$	write	yes	OUTPUT	NO (put it as private)
p	13	i	write	14	i	read	no	FLOW	NO (within same thread)
ic	14	i	write	14	$i+1$	read	yes	FLOW	**YES** (depends on previous iteration)

- The *ic* variable is the most critical, similar to *ec* previously.
- We parallelized the outer loop.
- Solution: precompute its values.

```
1  int ia, ic = 0;
2
3  int ib;
4  double u, v;
5  Point p;
6
7  for (ia = 1; ia <= nm - 2; ++ia) {
8      for (ib = 1; ib <= nm - 1 - ia; ++ib) {
9
10         u = (double) ia / nm;
11         v = (double) ib / nm;
12
13         p = tri_interp(A, B, C, u, v);
14         inter[ic++] = p;
15     }
16 }
```

DATA DEPENDENCY

COMPUTATION OF DISPLACEMENTS AND OFFSETS

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
indexOfProcess	14	i	read	15	i	write	no	ANTI	NO (within same thread)
indexOfProcess	15	i	write	14	$i+1$	read	yes	FLOW	**YES** (depends on previous iteration)
indexOfProcess	15	i	write	15	$i+1$	write	yes	OUTPUT	**YES** (depends on previous iteration)

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
sendcounts[i]	24	i	read	24	i	write	no	ANTI	NO (within same thread)
sendcounts[i]	24	i	write	25	$i+1$	read	yes	FLOW	**YES**
displs[$i-1$]	25	i	read	25	$i+1$	write	yes	ANTI	**YES**

- Too much overhead for so simple and bounded for loops.
- Not parallelized in the project.

DATA DEPENDENCY

LOCAL COMPUTATION OF PARTIAL SUMS

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
local_sum	5	i	write	5	$i+1$	read	yes	FLOW	NO (use REDUCE clause)

- The *local_sum* variable is accumulating all the sums contribution.
- Solution: use the reduction clause of OpenMP.

```
int local_sum = 0;

//Original code
for (int indx = 0; indx < local_length; indx++) {
    local_sum += f(local_E[indx].x, local_E[indx].y);
}

//Parallelized code
#pragma omp parallel for reduction(+ : local_sum)
for (int indx = 0; indx < local_length; indx++) {
    local_sum += f(local_E[indx].x, local_E[indx].y);
}
```

DATA DEPENDENCY

RICHARDSON EXTRAPOLATION

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
$R[m][k]$	3	m, k	write	3	$m, k+1$	read	yes	FLOW	**YES**
$R[m][k]$	3	m, k	write	3	$m+1, k+1$	read	yes	FLOW	**YES**

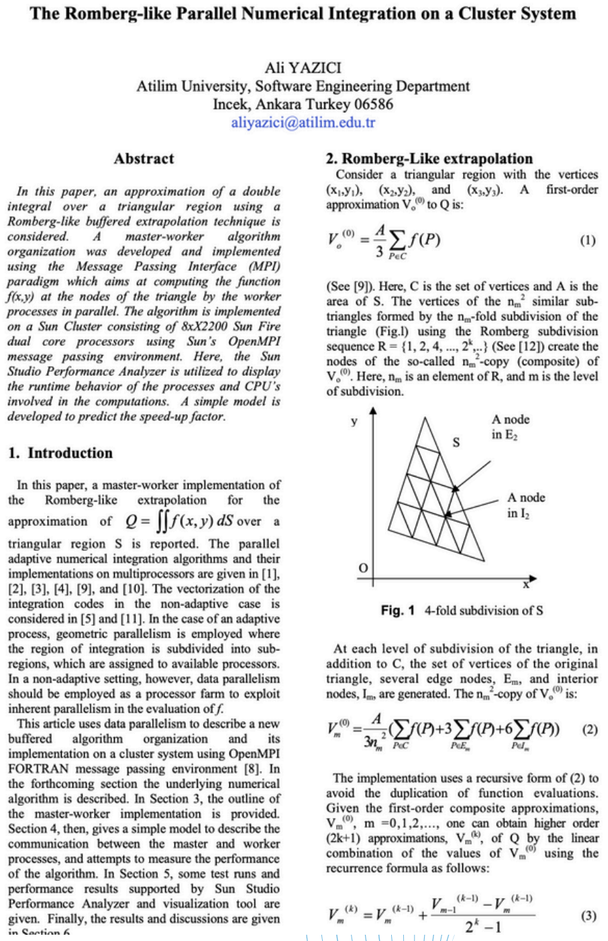
- There is a loop-carried flow data dependency (concurrence).
- Can be parallelized with parallel scan techniques.

```
1 for (int m = 1; m < MaxLevel; m++) {
2     for (int k = 1; k <= m; k++) {
3         R[m][k] = (pow(2, k) * R[m][k - 1]
4         - R[m - 1][k - 1]) / (pow(2, k) - 1);
5     }
6 }
```

THE MASTER-WORKER ALGORITHM

"The Romberg-like Parallel Numerical Integration on a Cluster System"

- **Author**: Ali Yazici (Atilim University).
- **Core Concept**: Solving the double integral over a triangle using a distributed Master-Worker model.
- **Key Innovation**: Uses buffered communication to counteract network latency in cluster systems.



The Romberg-like Parallel Numerical Integration on a Cluster System

Ali YAZICI
Atilim University, Software Engineering Department
Incek, Ankara Turkey 06586
aliyazici@atilim.edu.tr

Abstract

In this paper, an approximation of a double integral over a triangular region using a Romberg-like buffered extrapolation technique is considered. A master-worker algorithm organization was developed and implemented using the Message Passing Interface (MPI) paradigm which aims at computing the function f(x,y) at the nodes of the triangle by the worker processes in parallel. The algorithm is implemented on a Sun Cluster consisting of 8xX2200 Sun Fire dual core processors using Sun's OpenMPI message passing environment. Here, the Sun Studio Performance Analyzer is utilized to display the runtime behavior of the processes and CPU's involved in the computations. A simple model is developed to predict the speed-up factor.

1. Introduction

In this paper, a master-worker implementation of the Romberg-like extrapolation for the approximation of $Q = \iint f(x,y)\,dS$ over a triangular region S is reported. The parallel adaptive numerical integration algorithms and their implementations on multiprocessors are given in [1], [2], [3], [4], [9], and [10]. The vectorization of the integration codes in the non-adaptive case is considered in [5] and [11]. In the case of an adaptive process, geometric parallelism is employed where the region of integration is subdivided into sub-regions, which are assigned to available processors. In a non-adaptive setting, however, data parallelism should be employed as a processor farm to exploit inherent parallelism in the evaluation of *f*.

This article uses data parallelism to describe a new buffered algorithm organization and its implementation on a cluster system using OpenMPI FORTRAN message passing environment [8]. In the forthcoming section the underlying numerical algorithm is described. In Section 3, the outline of the master-worker implementation is provided. Section 4, then, gives a simple model to describe the communication between the master and worker processes, and attempts to measure the performance of the algorithm. In Section 5, some test runs and performance results supported by Sun Studio Performance Analyzer and visualization tool are given. Finally, the results and discussions are given in Section 6.

2. Romberg-Like extrapolation

Consider a triangular region with the vertices (x_1,y_1), (x_2,y_2), and (x_3,y_3). A first-order approximation $V_o^{(0)}$ to Q is:

$$V_o^{(0)} = \frac{A}{3}\sum_{P\in C} f(P) \qquad (1)$$

(See [9]). Here, C is the set of vertices and A is the area of S. The vertices of the n_m^2 similar sub-triangles formed by the n_m-fold subdivision of the triangle (Fig.l) using the Romberg subdivision sequence R = {1, 2, 4, ..., 2^k,..} (See [12]) create the nodes of the so-called n_m^2-copy (composite) of $V_o^{(0)}$. Here, n_m is an element of R, and m is the level of subdivision.

Fig. 1 4-fold subdivision of S

At each level of subdivision of the triangle, in addition to C, the set of vertices of the original triangle, several edge nodes, E_m, and interior nodes, I_m, are generated. The n_m^2-copy of $V_o^{(0)}$ is:

$$V_m^{(0)} = \frac{A}{3n_m^2}\left(\sum_{P\in C} f(P) + 3\sum_{P\in E_m} f(P) + 6\sum_{P\in I_m} f(P)\right) \qquad (2)$$

The implementation uses a recursive form of (2) to avoid the duplication of function evaluations. Given the first-order composite approximations, $V_m^{(0)}$, m =0,1,2,..., one can obtain higher order (2k+1) approximations, $V_m^{(k)}$, of Q by the linear combination of the values of $V_m^{(0)}$ using the recurrence formula as follows:

$$V_m^{(k)} = V_m^{(k-1)} + \frac{V_{m-1}^{(k-1)} - V_m^{(k-1)}}{2^k - 1} \qquad (3)$$

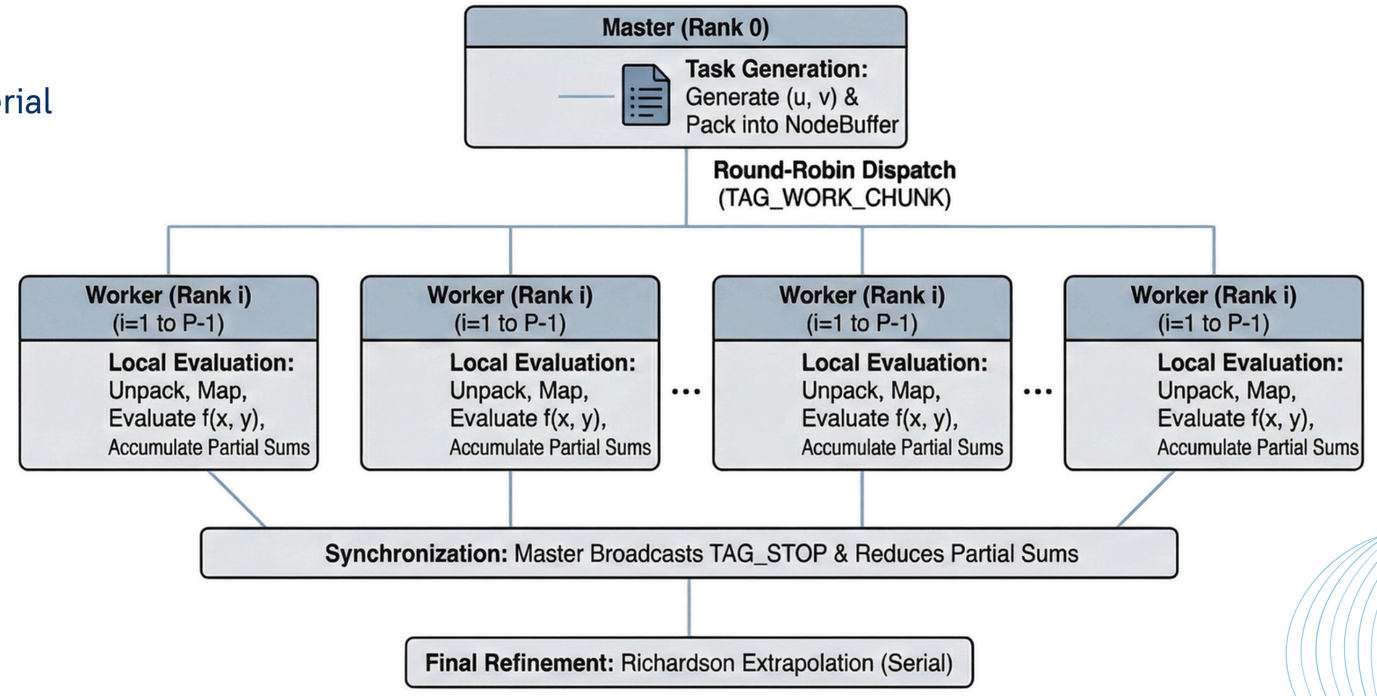
THE MASTER-WORKER ALGORITHM

Master (Rank 0): The Coordinator.

- Iterates over the 2D grid for the current refinement level
- Packs coordinates (u, v) into a NodeBuffer to reduce latency
- Sends full buffers to workers (Round-Robin)
- Issues TAG_STOP signals; performs serial Richardson Extrapolation

Workers(Rank 1 ... P-1): The Compute Units

- Waits for work chunks or stop signals
- Maps (u, v) -> (x, y) and evaluates f(x, y)
- Aggregates results into local partial sums (Edge vs. Interior)



DATA DEPENDENCY

MASTER LOOP (DISTRIBUTION)

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
buffer.u[]	10	j	write	10	$j+1$	write	yes	OUTPUT	NO (if count differs)
buffer.count	14	j	write	10	$j+1$	read	yes	FLOW	YES
buffer.count	21	j	write	10	$j+1$	read	yes	FLOW	YES
dest_worker	24	j	write	24	$j+X$	read	yes	FLOW	YES

- Strict Flow Dependency on buffer.count and dest_worker
- We cannot calculate which worker gets the next point without knowing if the current buffer is full
- The distribution loop must remain serial.

```
1  NodeBuffer buffer;
2  buffer.count = 0;
3  int dest_worker = 1;
4
5  for (long long i = 0; i <= nm; i++) {
6      for (long long j = 0; j <= nm - i; j++) {
7          // ... (skip logic omitted) ...
8
9          // 1. Write to buffer at current index
10         buffer.u[buffer.count] = (double) i / nm;
11         buffer.v[buffer.count] = (double) j / nm;
12
13         // 2. Increment Counter
14         buffer.count++;
15
16         // 3. Check Condition
17         if (buffer.count == BUFFER_SIZE) {
18             // MOCK_MPI_Send(&buffer, dest_worker);
19
20             // 4. Reset Counter
21             buffer.count = 0;
22
23             // 5. Update Worker ID
24             dest_worker++;
25             if (dest_worker >= size) dest_worker = 1;
26         }
27     }
28 }
```

DATA DEPENDENCY

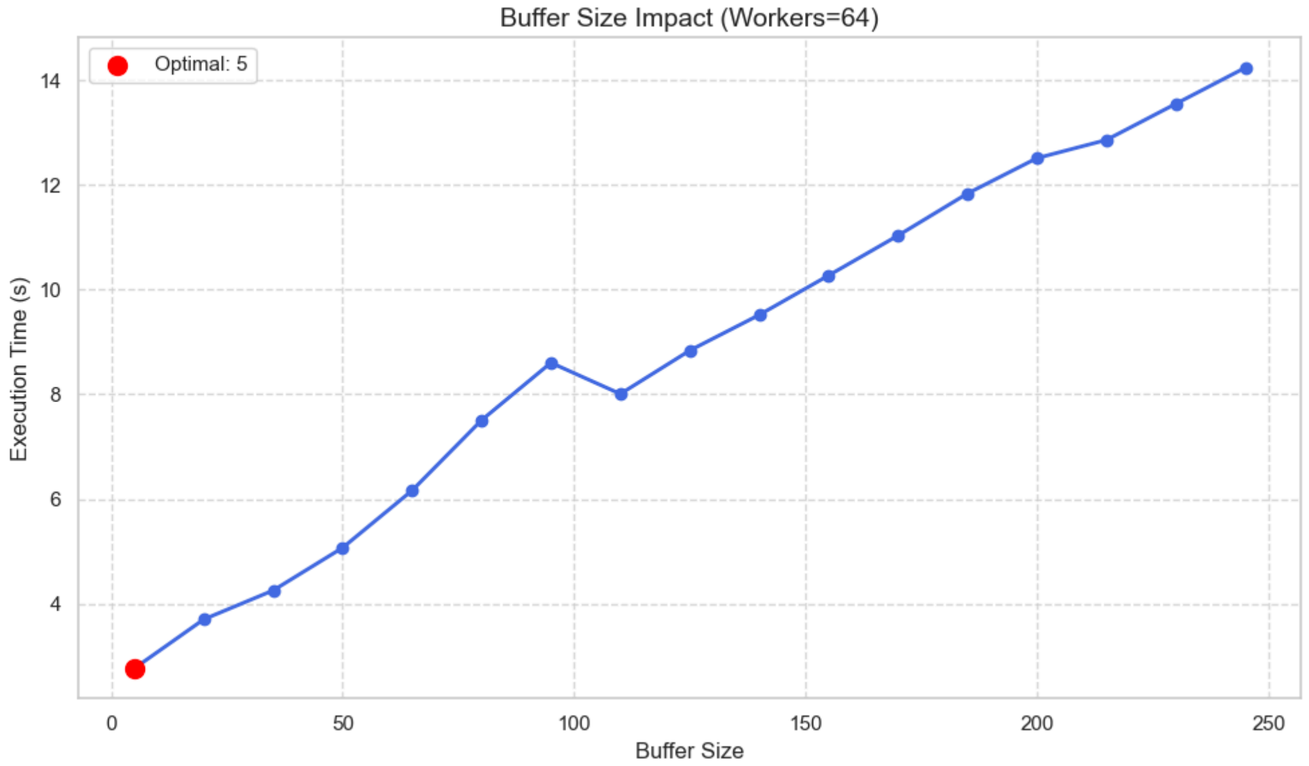
WORKER COMPUTATION LOOP

Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of dataflow	Issue
	Line	Iter.	Access	Line	Iter.	Access			
`local_sum`	8	k	write	8	$k+1$	read	yes	FLOW	NO (Reduction)

- Reduction on local_sum
- Theoretically parallelizable (OpenMP), but not implemented

```
1 double local_sum = 0.0;
2 // ... (receive buffer) ...
3
4 for (k = 0; k < received_count; k++) {
5     double val = u[k] * 2.0; // Function eval
6
7     // Accumulate result
8     local_sum += val;
9 }
```

BUFFER SIZE ANALYSIS



Setup:

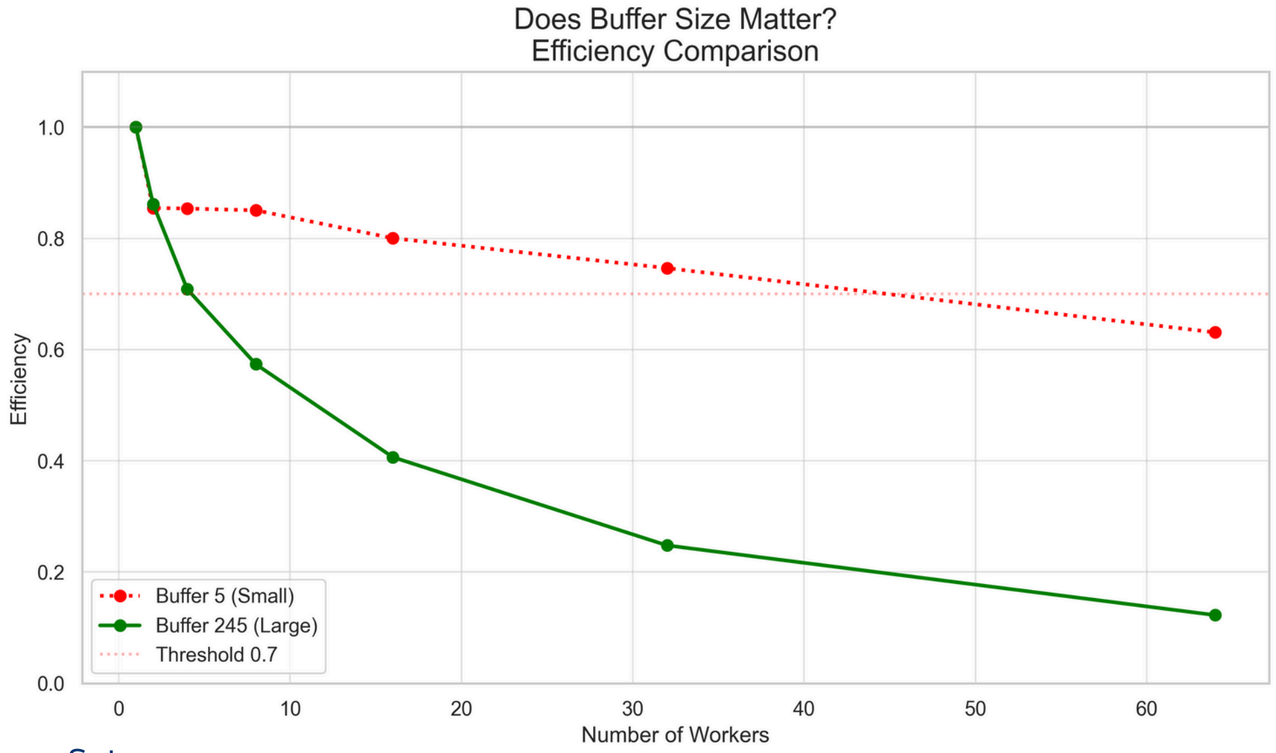
- increasing MPI processes, 1 000 000 problem size.
- Configuration standard.

Hypothesis vs. Reality

- **Paper Suggestion**: "Buffer size has minimal impact"
- **Our Finding**: Significant performance degradation with larger buffers
- Monotonic increase in execution time as buffer size grows.

BUFFER SIZE ANALYSIS

WHY EFFICIENCY DROPS?



Setup:

- increasing MPI processes, 1 000 000 problem size.
- Configuration standard.

1. Master Bottleneck (First-Byte Latency):

- Large B forces the Master to compute many points serially before sending the first chunk.
- Workers sit idle (Starvation) during the "filling phase."

2. Granularity & Load Imbalance:

- Large B = Fewer total chunks.
- One worker gets a huge final chunk while 63 others wait.

3. MPI Constraints:

- Higher serialization and memory copying costs for larger messages.

BENCHMARKING

- Environment: UNITN HPC2 CLUSTER.
- All configurations (MPI processes - OpenMP threads per process - problem size) have been executed 5 times. Then we took as execution time the **minimum** time.

What do we mean with “problem size”?

- Heavier part is function evaluation, so we consider as problem size the *computational complexity* of the integrand function.

PLACEMENT ANALYSIS

- Configuration 1: ***scatter:excl***
- Configuration 2: no placement constraints.

Size	Std dev with config 1	Std dev with config 2
250 000	0.013880	0.135199
500 000	0.026283	0.070129
1 000 000	0.056074	0.152946
2 000 000	0.024530	0.287427
4 000 000	0.071851	0.734122
8 000 000	0.384598	3.999597

Result: less standard deviation between measurements and overall lower execution times.

ALGORITHM #1

OPENMP SCALABILITY

Threads	Time (s)	Speedup	Efficiency
1	170.992	1.000	1.000
2	86.867	1.968	0.984
4	44.589	3.835	0.959
6	30.199	5.662	0.944
8	23.595	7.247	0.906
10	19.012	8.994	0.899
12	15.838	10.797	0.900
14	13.850	12.346	0.882
16	12.287	13.916	0.870

Results:

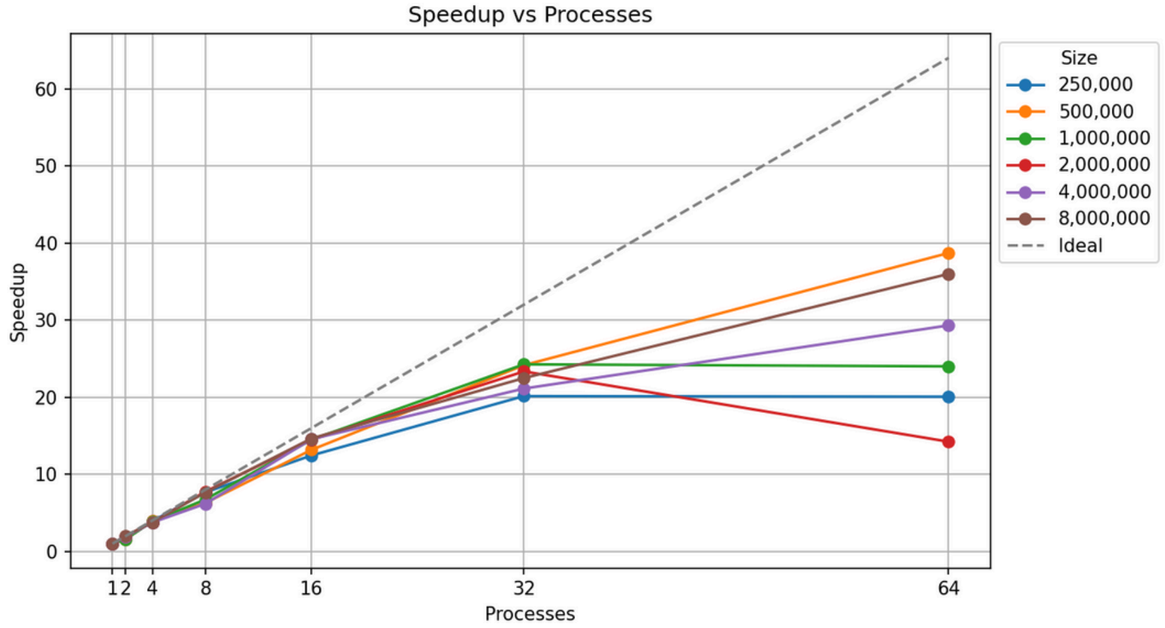
- strongly scalable shared-memory parallelization
- computation “wins” over communication overhead

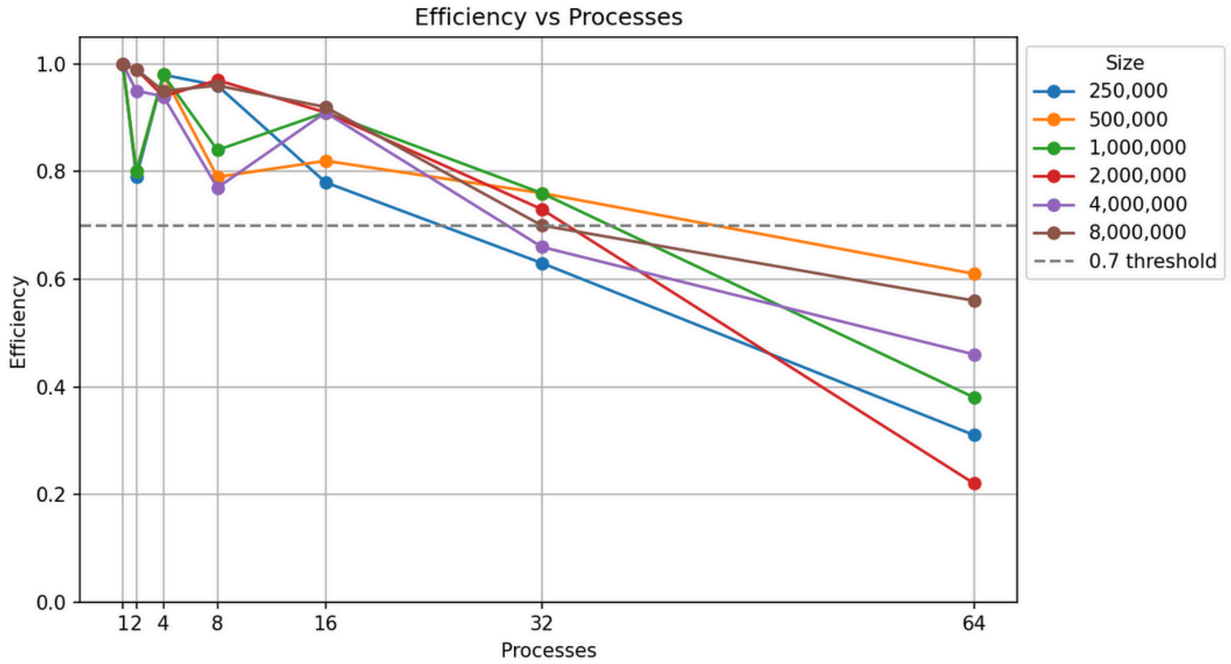
Setup:

- 1 MPI process, increasing threads, 1 000 000 problem size.
- Configuration ***pack:excl***.

ALGORITHM #1

MPI (STRONG) SCALABILITY





ALGORITHM #1

MPI (STRONG) SCALABILITY

Size	p	1	2	4	8	16	32	64
250,000	S	1.0	1.58	3.92	7.66	12.45	20.14	20.07
	E	1.0	0.79	0.98	0.96	0.78	0.63	0.31
500 000	S	1.0	1.60	3.94	6.29	13.18	24.15	38.70
	E	1.0	0.80	0.98	0.79	0.82	0.76	0.61
1,000,000	S	1.0	1.60	3.91	6.73	14.50	24.26	24.00
	E	1.0	0.80	0.98	0.84	0.91	0.76	0.38
2 000 000	S	1.00	1.97	3.77	7.75	14.48	23.34	14.23
	E	1.00	0.99	0.94	0.97	0.91	0.73	0.22
4 000 000	S	1.00	1.90	3.75	6.18	14.56	21.12	29.31
	E	1.00	0.95	0.94	0.77	0.91	0.66	0.46
8,000,000	S	1.0	1.99	3.78	7.65	14.65	22.48	35.99
	E	1.0	0.99	0.95	0.96	0.92	0.70	0.56

Results:

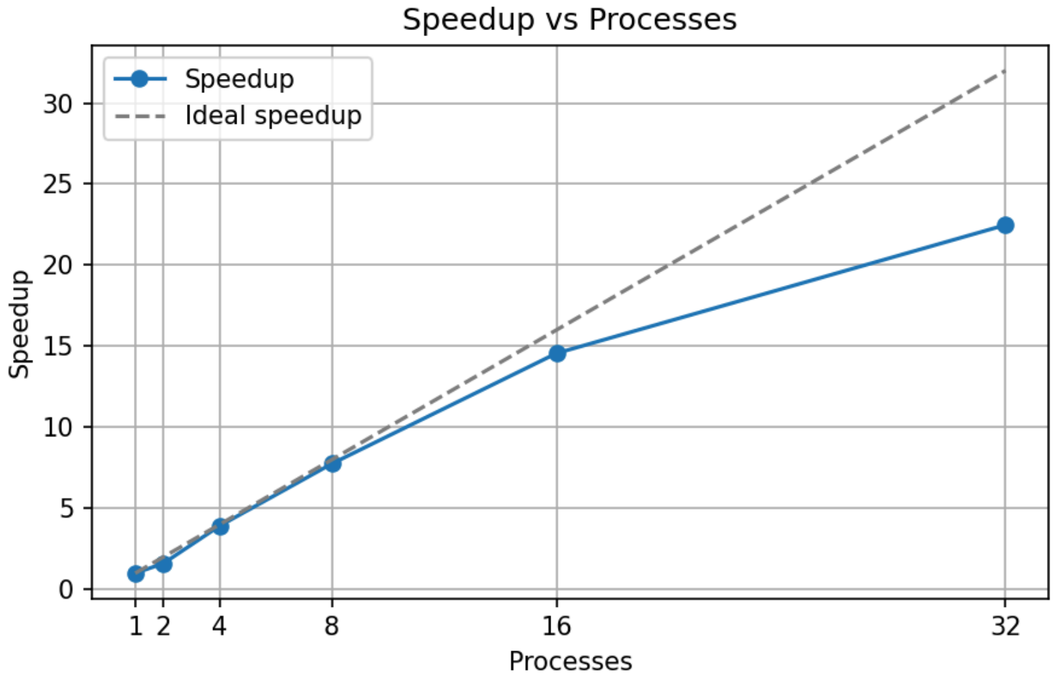
- scalable up to **16 processes** even with *small sizes*
- positive efficiency even with **32 processes** with *medium-big sizes*
- intrinsic limitation around **64 processes**

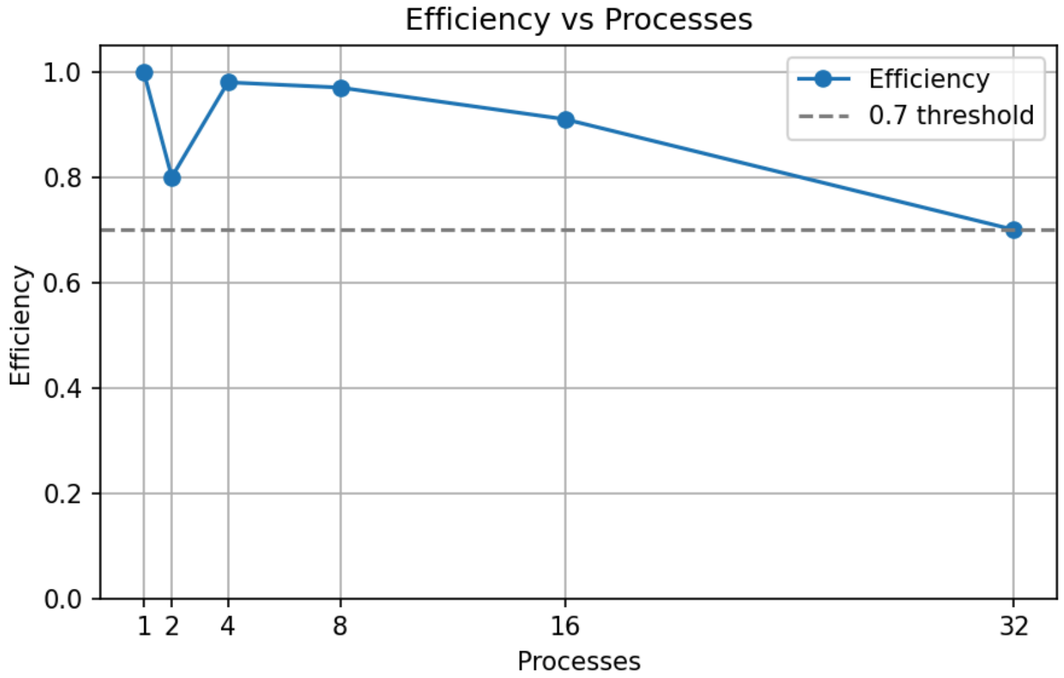
Setup:

- increasing MPI process, 4 threads each, increasing problem size.
- Configuration ***scatter:excl*** (excluding 64 scenario).

ALGORITHM #1

MPI (WEAK) SCALABILITY





ALGORITHM #1

MPI (WEAK) SCALABILITY

Size	Proccssscs	Time (s)	Speedup	Efficiency
250 000	1	10.59	1.00	1.00
500 000	2	13.45	1.60	0.80
1 000 000	4	10.98	3.91	0.98
2 000 000	8	11.09	7.75	0.97
4 000 000	16	11.64	14.56	0.91
8 000 000	32	15.13	22.48	0.70

Setup:

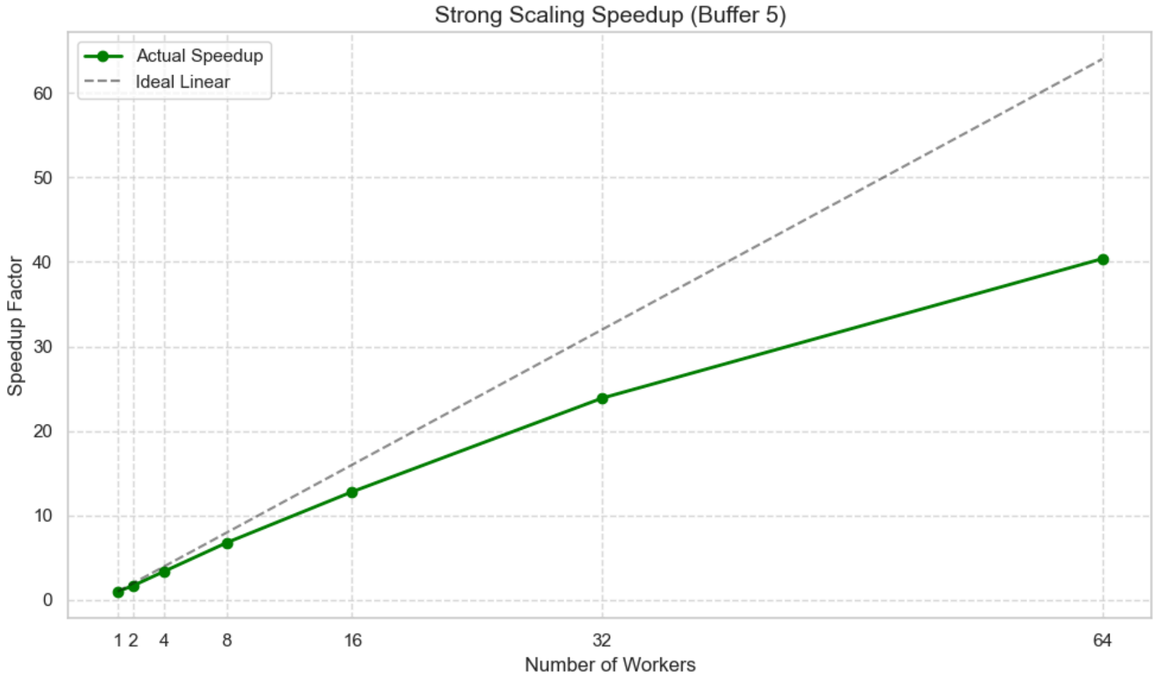
- increasing MPI process, 4 threads each, increasing problem size proportionally.
- Configuration ***scatter:excl***.

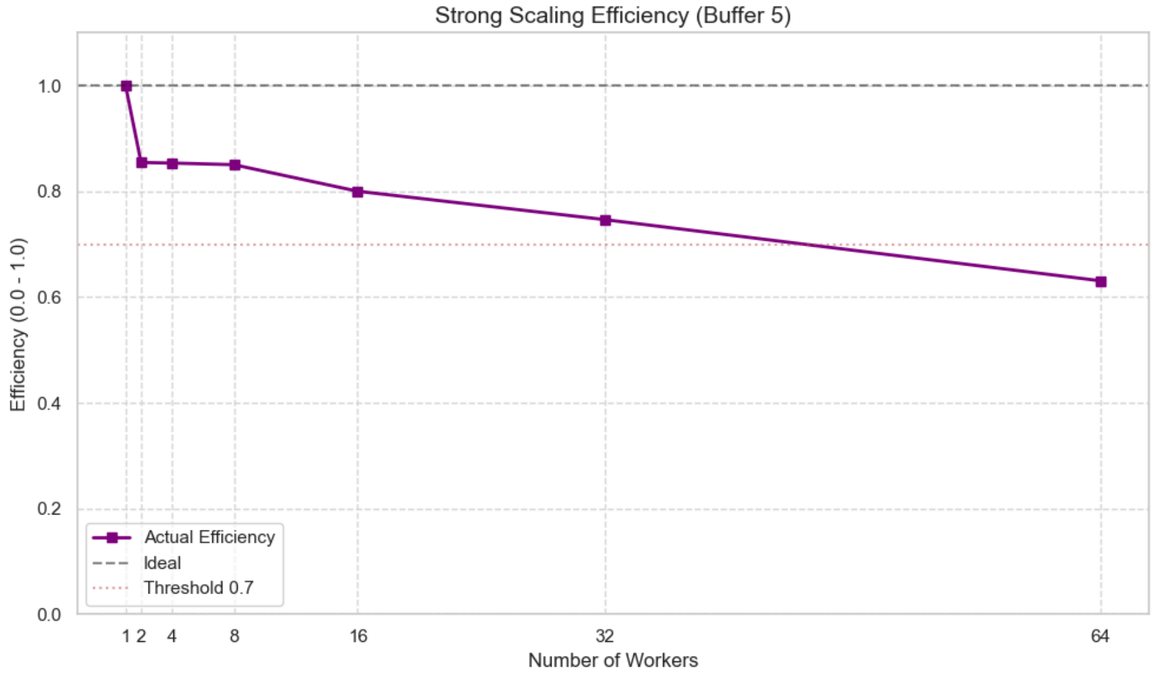
Results:

- constantly high efficiency when increasing significant the problem size
- execution times very stable when processes follow proportionally the problem size

THE MASTER-WORKER ALGORITHM

STRONG SCALABILITY



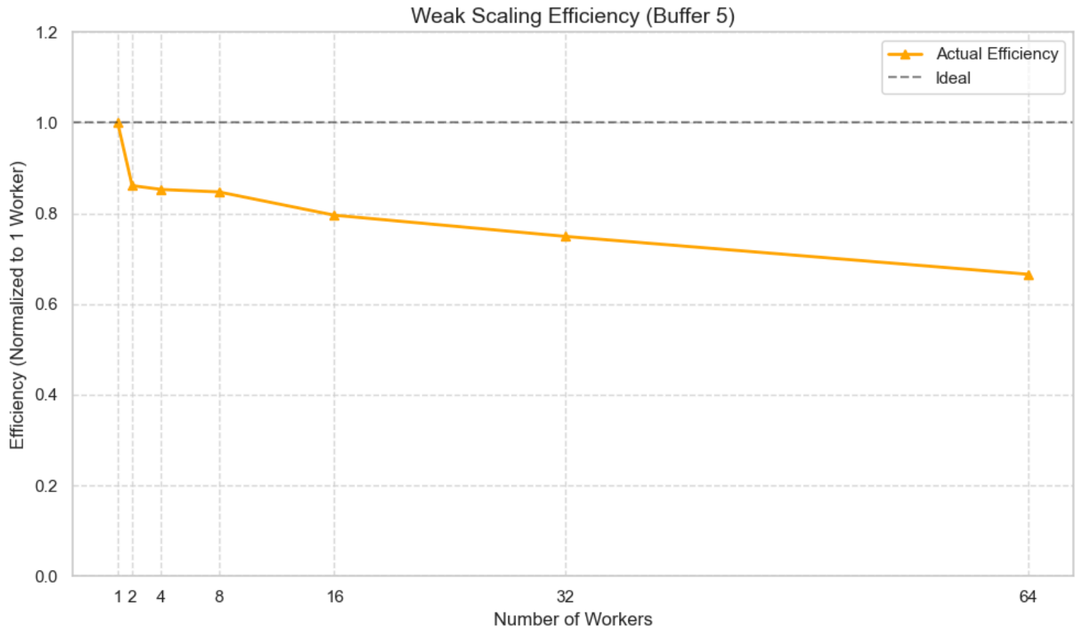


- Achieves almost perfect linear speedup up to 32 workers (S ~ 23.9x)
- At 64 workers, speedup reaches ~40x (deviation from ideal 64x)
- The problem size 1.000.000 becomes too small for 64 processors

- Consistently maintains E > 0.8 for up to 16 workers
- At 64 workers, efficiency is “almost acceptable” at ~0.63
- The slight dip at W=64 reflects the serial overhead of the Master managing communication for such a short task.

THE MASTER-WORKER ALGORITHM

WEAK SCALABILITY



Buffer Size	Problem Size	Workers	Time (s)	Speedup	Efficiency
5	250 000	1	27.91	1.00	1.00
	500 000	2	32.40	1.72	0.86
	1 000 000	4	32.73	3.41	0.85
	2 000 000	8	32.94	6.78	0.85
	4 000 000	16	35.06	12.74	0.80
	8 000 000	32	37.25	23.98	0.75
	16 000 000	64	41.92	42.61	0.67
50	250 000	1	27.91	1.00	1.00
	500 000	2	32.21	1.73	0.87
	1 000 000	4	33.81	3.30	0.83
	2 000 000	8	35.32	6.32	0.79
	4 000 000	16	40.90	10.92	0.68
	8 000 000	32	51.74	17.26	0.54
	16 000 000	64	78.05	22.89	0.36
95	250 000	1	27.91	1.00	1.00
	500 000	2	32.14	1.74	0.87
	1 000 000	4	34.22	3.26	0.82
	2 000 000	8	39.92	5.59	0.70
	4 000 000	16	53.45	8.35	0.52
	8 000 000	32	79.31	11.26	0.35
	16 000 000	64	137.75	12.97	0.20

- Scaled complexity (250.000 ops per worker). Problem size grows 64x (from W=1 to W=64)
- High efficiency maintained (0.67 at max scale)
- The minimal increase in time (only 1.5x slower for a 64x larger problem) proves low communication overhead
- Excellent weak scalability; suitable for massive workloads.

Setup:

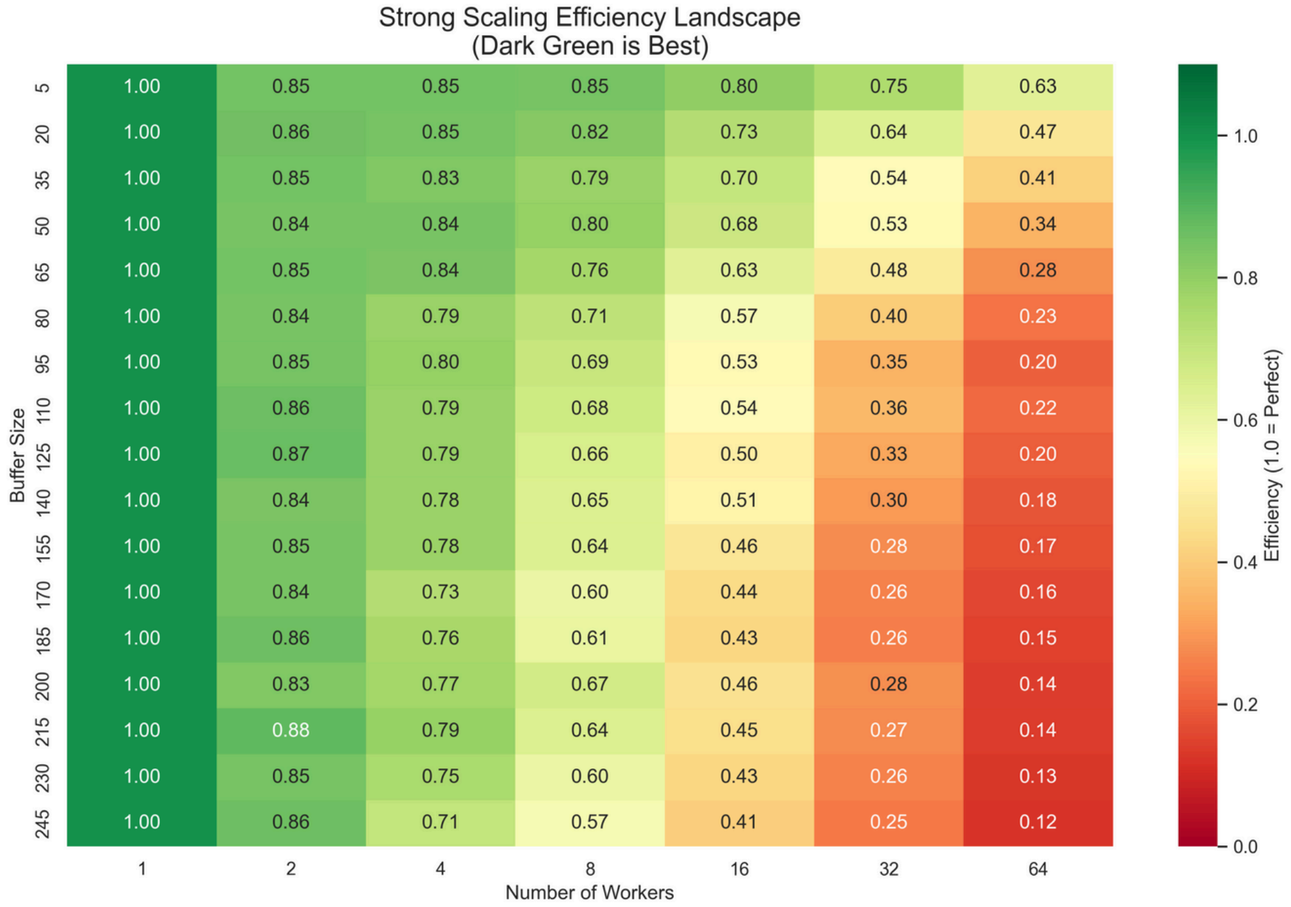
- increasing MPI processes, increasing problem size proportionally.
- Configuration standard.

THE MASTER-WORKER ALGORITHM

BUFFER SIZE IMPACT

Results:

- scalable up to **32 processes**
- also here communication dominates with **64 processes**
- high-efficincy zone with **small buffers**
- rapid degradation of efficiency with bigger buffer with many processes involved



Setup:

- increasing MPI process, 1 000 000 problem size.
- Configuration standard.

COMPARISON WITH REFERENCE PAPER

The "Unknown Variable"

- Direct comparison is impossible (Paper does not define computational complexity of "Size")

Contrast 1: Buffer Size

- Paper: "Buffer size is not important." (Likely used a heavier kernel where communications were negligible).
- Our Result: Buffer size is critical. Efficiency drops dramatically with large buffers

Contrast 2: Scalability

- Paper: Poor strong scalability; performance degrades quickly at 8-16 processes for small sizes. Peak efficiency 0.88 at 16 processes (large size).
- Our Result: Superior scalability. We maintain good results at smaller sizes and scale effectively to 32+ processes.

CONCLUSION

Scalability Parity

- Both scale well up to 16 processes
- Algorithm #2 (Buffer-Based): Maintains better efficiency at 32 MPI processes

Sensitivity

- Algorithm #1 (Hybrid): Needs larger problem sizes to scale beyond 16 processes
- Algorithm #2: More robust at smaller sizes due to dynamic distribution

Verdict

- Both algorithms can be considered well-designed.
- Algorithm #1 can be used to solve massive computational problem thanks to its weakly scalabiltiy capabilities (efficiency >= 0.9 with 16 processes and 4 000 000 of size).
- Algorithm #2 can reach good efficiency even with small problem sizes.

THANK YOU