Parallel numerical integration using Romberg's method: a comparison of two different techniques

January, 2026

Gabriele Volani
Dept. of Information Engineering and Computer Science
University of Trento
Trento, Italy
gabriele.volani@studenti.unitn.it

Sanasar Hamburdzamyan
Dept. of Information Engineering and Computer Science
University of Trento
Trento, Italy
s.hamburdzamyan@studenti.unitn.it

Abstract

This paper examines the use of the Romberg's method to compute the double integral over a triangular region. The problem was inspired by another relevant work in the field.
Two algorithms have been studied and compared. We have designed and implemented an ad-hoc parallelization strategy using MPI and OpenMP and we have implemented a master-worker buffer-based algorithm that relies on MPI.
Finally, both implementations were run on the UNITN HPC2 cluster to study their performance (strong and weak scalability).

***Index Terms*—numerical integration, Romberg's method, MPI, OpenMP, benchmarking**

Project repository — https://github.com/gabri17/HPC-Project.git

1. Introduction

Mathematically, numerical integration represents a family of algorithms for calculating (approximately) the numerical value of a definite integral. Numerical integration techniques are used in real-world scenarios (like engineering, mechanics and physics) when an exact and closed-form anti-derivative of the integrand function $f(x)$ cannot be found or it is known but then it is much more feasible to compute a numerical approximation of the integral rather than computing the anti-derivative.

One of the techniques used for numerical integration is the **Romberg's method**, which improves accuracy by repeatedly applying Richardson Extrapolation to the trapezoidal rule. Although the trapezoidal rule is simple, its error behave in a well-known way: Romberg's method exploits this behavior to achieve accurate (as much as wanted) estimations faster than with any other numerical integration techniques. The method works recursively by building a triangular table and can be divided into two steps: at the first step we compute the integral using the standard trapezoidal rule by doubling at each time the number of sub-intervals of the integration interval; secondly, we apply a Richardson extrapolation: we do not take the finest trapezoidal estimate, but we combine the previous estimates to generate more accurate estimations following a specific formula.

In appendix A it is possible to find the formula of the Richardson extrapolation and a detailed explanation of the method.

2. Problem definition

The problem we address for parallelization is about the approximation of a double integral over a triangular region S using a Romberg-like technique:

$$Q = \int\int f(x,y)dS$$

We consider the triangular region S with set of vertices C and area A. The idea is to build a triangular table, like the Romberg one, by applying an extrapolation at multiple levels to refine the approximation Q until a certain accuracy.

The first approximation of Q is:

$$R[0][0] = \frac{A}{3} \sum_{(x,y)\in C} f(x,y)$$

As for the general Romberg method, we first have to compute the first column of the table.

The idea is to add new vertices at each iteration by finding the midpoints of the sides that bound the triangle.

Initially, at $n = 0$, we only have the vertices of the triangle. At the next iteration, we find the midpoints of the triangle sides and obtain a total of 6 points: the 3 vertices and 3 points called "edge points" (points that lie on the sides, part of the set E_n). We connect the edge points (by letting pass through any point two lines parallel to the other two sides) and divide the triangle into 4 smaller triangles. At the third iteration, with $n = 2$, we have 16 small triangles because for each side delimited by two points on the edges of the triangle, we find the midpoints (so another 6 edge points). We then connect the points as above and for the first time we also have 3 points that lie inside the triangle. These are interior points (set I_n).

A picture in appendix B will clarify the overall subdivision mechanism.

At each step n, the mathematical part consists in calculating the sets I_n and E_n (of interior and edge points). Then, it is necessary to apply the following formula to obtain a first estimate of the integral with that level n of subdivision.

$$\begin{aligned} R[n][0] = \frac{A}{3 \cdot 4^n} \Bigg(& \sum_{(x,y)\in C} f(x,y) \\ & + \sum_{(x,y)\in E_n} f(x,y) \\ & + \sum_{(x,y)\in I_n} f(x,y) \Bigg) \end{aligned} \tag{1}$$

The second step of our Romberg-like procedure consists to apply a Richardson extrapolation for increasing the accuracy of the estimations.

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

The solution of the problem is basically a Romberg technique, adapted to deal with the specific type of input we have.

2.1. Methodology

Our work is inspired on the related work [1]: here the problem is described and the general solution with a Romberg technique is formalized.

Additionally there is a pseudo-code algorithm proposed to solve the problem with a master-worker technique that uses the Message Passing Interface (MPI) paradigm.

Our project is mainly divided into two parts.

The first part consists in the implementation of the pseudo-code provided in the paper, while the second one is the design and the implementation of a different parallel strategy to solve the problem.

After that, we have analyzed the performances of both implementations and compared the efficiency and the scalability across increasing problem size and number of processes used.

3. Our algorithm

We have analyzed the structure and the mathematics behind the problem. The overall complexity relies on the first step (equation (1)), where lots of evaluations of the function must be done: in fact, this integration problem is particularly complex when the function f to integrate is very computationally expensive.

In the table I we show how the number of edge and interior points increasea while increasing the step. With $n = 7$, 8 382 functions evaluations must be done. A very huge number if the function is computationally heavy. For that reasons, we have focused on *parallelize the first step* of the algorithm. The second step (the Richardson extrapolation) is not so complex: it just consists in few algebraic operations across already computed values in the Romberg matrix and, since it is a matrix up to *8x8*, its computational complexity is much more feasible compared to the function evaluations.

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	$3 \cdot 2^n - 3$	$\frac{(2^n+1)(2^n+2)}{2} - 3 \cdot 2^n$

From empirical observations, we agreed to set the refinement level to 8, since it allows to give enough accurate estimations of the integrand.

3.1. Parallel design

The idea is to apply a hybrid strategy involving both MPI and OpenMP parallelization. Since the performance bottleneck of the problem is at (1) we focused on parallelize that part. The (2) is performed serially by the master process (MPI process 0).

First the master process is entitled to compute the interior and edge points for each level n. Then both sets are distributed equally across all processes with a `MPI_Scatterv`.

After that, each process (including the master one) is entitled to evaluate the function on the smaller subset of points. This is the crucial part: respect to the serial implementation, we do not have only one process that must evaluate the function to both the set E_n and I_n, but each process will take care of a smaller subset of them. Functions evaluation and partial sums can be performed independently and then combine together at a single final step.

OpenMP parallelization is used both in the master process for speeding up the generation of edge and interior points and in all the processes to parallelize the serial evaluations of the function on the smaller subset of edge and interior points assigned to them. Using few threads per process we can reduce again the workload and parallelize even the local computations of the function and of the partial sum.

Finally a `MPI_Reduce` is used to combine together all the partial sums contribution by all the processes. At the master process the final value for the cell $R[n][0]$ of the table is obtained. After that, Richardson extrapolation is performed by the master to complete the table.

3.2. Data dependency analysis

After having analyzed and implemented the parallelization using MPI (where we distributed the workload using `MPI_Scatterv` and `MPI_Reduce`), we started to analyze the code for a shared-memory parallelization using OpenMP.

There are still several serial sections in the code:

- the generation function of master process: the MPI process 0 is in fact entitled of generating edge and interior nodes for each level of subdivision. This part is done serially, but there are loops we can optimize both in generation of edge and interior points;
- the subdivision of points to distribute: master process is also entitled to, once having generated all the points, distribute them across the different MPI processes. Also this part is done serially but could be parallelized;
- the local evaluation of the integrand function on the edge and interior nodes: this is the most heavy part that is done serial by all the process. It consists in evaluating the integrand function and computing partial sums on the subset of edge and interior nodes assigned.

Before starting to adapt immediately these parts for an OpenMP parallelization, we have performed a data dependency analysis. In fact, here we are dealing within a shared memory context and it is important to avoid situations in which multiple threads access and modify the same memory region.

All data dependence tables are shown in appendix C.

3.2.1. Edge and interior points generation

In the case of edge points generation, the serial code was the following:

```
int t, ec=0;
double u;

Point pAB, pBC, pCA;
for (t = 1; t <= nm - 1; ++t) {
    // Contribution of each point
    u = (double) t / nm;

    // Each point on the edge
    // is a linear interpolation of its extremes
    pAB = lerp(A, B, u);
    edge[ec++] = pAB;

    pBC = lerp(B, C, u);
    edge[ec++] = pBC;

    pCA = lerp(C, A, u);
    edge[ec++] = pCA;
}
```

Data dependencies are shown in appendix C.1 at table VI.

It is important to underline that the *dynamic extent* of this routine includes the function `Point lerp(Point P1, Point P2, double t).`

In this function two vertices of the triangle region are passed: this could lead to potential dependency problems in case that the logic of the routine is modifying the points (since they are shared among all threads). Nevertheless, this is not the case since the `lerp` function is only doing the linear interpolation of that two points passed without modifying the parameters, so it does not give any problems.

There are plenty of data dependencies in the code that should be addressed. For variables `u, pAB, pBC, pCA` we have no critical issues: it is true that they are shared, but we can specify a private scope or just declare them inside the loop (since their lifetime is limited to loop scope).

It is important to observe that even if it is true that `edge` is a shared array and is accessed at the same time by all the threads contemporary, it does not give problem since all threads access different memory area of it.

Variables `ec` is instead more problematic: this memory location is shared among all threads and it is accessed by all of them both in writing and in reading. Furthermore, the value of it depends on the previous iterations, since it behaves like a counter. If we want to parallelize that code, we must pre-compute all the values for *ec* for all iterations. The code is available in appendix C.1.

By doing so, every thread is operating on its local copy of `ec` thanks to a lightweight loop that is pre-computing all its values.

In appendix C.1 at table VII we can find the dependency table for the loop generating interior points.

In interior nodes generation, the situation is a little bit different, we have two nested loops for creating the points:

```
int ia, ic = 0;

int ib;
double u, v;
Point p;

for (ia = 1; ia <= nm - 2; ++ia) {
    for (ib = 1; ib <= nm - 1 - ia; ++ib) {

        u = (double) ia / nm;
        v = (double) ib / nm;

        p = tri_interp(A, B, C, u, v);
        inter[ic++] = p;
    }
}
```

Our decision is to parallelize the *outer* loop and keep the inner one serial. After making this decision, we can see that situation is similar as above: most of variables can be put private or just declared inside the loop where they are used, to avoid dependency issues; also here the function of dynamic extent `tri_interp` does not give problem since it is not modifying its parameters. However, for `ic` we observe a situation similar to that of `ec`. To fix this issue, we have precomputed also it. The code is available in appendix C.1.

Inevitably, these choices introduce overhead in terms of space used, but they allow at least a slight improvement in performance by parallelizing the generation of points, which can be even several thousand in the highest refinement levels, as seen in table I.

3.2.2. Distribution of points

The function examined now is the one computing the `sendcounts` and `displs` for the `MPI_Scatterv`. The code and the data dependencies tables are provided in appendix C.2. In that case, parallelization would be disproportionate to the computation cost, given that the loops analyzed are bounded by the number of processes (at most 64) and involve only basic operations. So, a parallel OpenMP code is provided as example, but in the project the logic is kept serial.

3.2.3. Local computation of partial sums

This is the most useful parallelization. Every MPI process has to do a loop (actually two, one for edge and one for interior points assigned to it) for evaluating the function on the points it has to manage and compute partial sums. This is originally done serially, but can be very easily parallelized with OpenMP.

Here we show the loop before and after the parallelization, while data dependency table is in appendix C.3, at table X:

```
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);
}
```

The only issue is that `local_sum` is an accumulator, accessed by every thread. Anyway, this problem is easily solved by the OpenMP `reduction` clause.

This parallelization is the most important because it allows to reduce the execution time significantly, since functions evaluations represent the most heavy parts of the algorithm.

3.2.4. Unparallelizable parts

Finally, we have also performed a data dependence analysis on the Richardson extrapolation part. Although we have already explained why we have decided to not parallelize this loop, we will highlight the difficulties that such parallelization would entail.

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

The main issue is the presence of a *recurrence* caused by a flow loop-carried data dependence, as shown in appendix C.4 at table XI. In principle, it could be parallelized using a parallel scan technique, but the overhead introduced would completely outweigh any potential performance benefits. In fact the loop only fills the entries of an 8x8 triangular matrix (so iterations are very limited) and operations performed are very simple, just additions and subtractions.

4. The Master-Worker Algorithm

The next part of our project focuses on the implementation of the algorithm presented (with pseudo-code) in [1], from which the addressed problem was also derived. This approach is designed specifically for distributed memory cluster systems and relies exclusively on the Message Passing Interface (MPI) standard.

The core strategy utilizes a **Master-Worker** paradigm with a manual buffering mechanism to handle the integration over a triangular region problem.

Role of the Master Process

The Master process (rank 0) does not perform the heavy lifting of function evaluation. Instead, it acts as a dynamic scheduler and coordinate generator. Its responsibilities are:

1) **Grid Traversal:** It iterates over the $2D$ grid defined by the current Romberg refinement level.
2) **Buffering:** Instead of sending a single coordinate pair (u, v) to a worker, which would incur massive network latency, the Master packs these coordinates into a `NodeBuffer` structure.
3) **Distribution:** Once a buffer is full (determined by `BUFFER_SIZE`), the Master dispatches it to the next available worker in a round-robin fashion using `MPI_Send`.
4) **Synchronization:** After generating all points for a level, it broadcasts a `TAG_STOP` signal to ensure all workers have finished before performing the Richardson extrapolation serially.

Role of the Worker Process

The Worker processes (Ranks $1 \ldots P-1$) act as pure computational units. Their lifecycle is a continuous loop:

1) **Receive:** Wait for a message. If it is a `TAG_STOP`, they pause for the next level. If it is a `TAG_WORK_CHUNK`, they proceed.
2) **Unpack & Compute:** They iterate through the received buffer, extracting (u, v) pairs, mapping them to the physical triangle (x, y), and computing the complex function $f(x, y)$.
3) **Accumulate:** Results are aggregated into local partial sums (Edge vs. Interior).

4.1. Data dependency analysis

Before implementing the algorithm, we performed a rigorous data dependency analysis on the source code. Even if OpenMP was practically not used in the implementation, this can help in future to understand why the Master process is strictly sequential and cannot be easily parallelized.

4.1.1. Master: Work Distribution Loop

The logic for generating points and filling the buffer is shown below. This code segment highlights the manual packing strategy.

```
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 }
```

The data dependency analysis for this loop is presented in the Appendix D Table XII.

The variable `buffer.count` presents a critical flow dependency. Additionally, line 21 (`buffer.count = 0`) resets this variable based on a conditional check. This means the state of `buffer.count` in iteration $j + 1$ is entirely dependent on the history of previous iterations.

Similarly, `dest_worker` creates a circular dependency to ensure round-robin distribution.

This loop is inherently serial. Parallelizing it would require complex prefix-scan operations to pre-calculate offsets, which is computationally expensive and unnecessary given that coordinate generation is lightweight.

4.1.2. Worker: Computation Loop

The worker receives a buffer and performs the reduction.

```
double local_sum = 0.0;
// ... (receive buffer) ...

for (k = 0; k < received_count; k++) {
    double val = u[k] * 2.0; // Function eval

    // Accumulate result
    local_sum += val;
}
```

The dependencies for the worker loop are shown in Appendix D, in Table XIII. The variable `local_sum` exhibits a loop-carried flow dependency. However, since addition is associative, this is a standard reduction pattern.

4.2. Buffer Size Analysis

The efficiency of the this algorithm relies heavily on the choice of the buffer size B (with buffer size we indicate the number of points in the buffer). The reference paper suggests that beyond a certain minimum, the buffer size has a minimal impact on the overall speedup factor. Ideally, one aims for a "sweet spot" where the buffer is large enough to amortize network latency (saturating bandwidth) but small enough to ensure a granular distribution of work.

However, our experimental results on the cluster reveal a distinct behavior that contradicts the reference paper's "negligible impact" claim. We conducted a parameter sweep for $B \in [5, 245]$ using the maximum available resources ($W = 64$ workers) to stress-test the distribution mechanism.

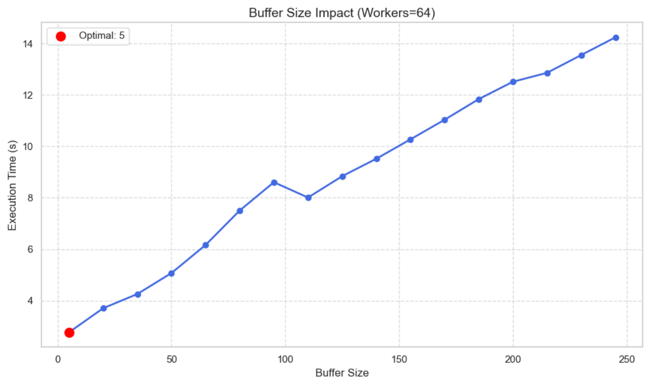

Fig. 1. Impact of Buffer Size on Execution Time ($W = 64$ and problem size 1 000 000). Contrary to the reference paper, we observe a linear degradation in performance as buffer size increases.

As illustrated in Figure 1, we observed a monotonic increase in execution time as the buffer size increased:

- **Small Buffers** ($B = 5$)**:** This configuration yielded the best performance, with a minimum execution time of **2.76 seconds**. The Master processes small chunks rapidly, keeping all 64 workers constantly fed with data.
- **Large Buffers** ($B = 245$)**:** This configuration resulted in an execution time of **14.23 seconds**—a performance degradation of over **5x**.

This behavior indicates two critical phenomena in our specific implementation:

1) **Master Generation Bottleneck:** The Master process generates coordinates (u, v) serially. When B is large (e.g., 245), the Master must compute 245 pairs before sending the first packet. During this "filling" phase, workers sit idle (Starvation). With $B = 5$, the "first byte latency" is minimized, allowing workers to start almost immediately.
2) **Load Imbalance (Granularity):** With 64 workers and a finite grid size, large buffers reduce the total number of "chunks" available. Towards the end of the computation, a large buffer might assign a heavy block of work to one worker while the other 63 sit idle, waiting for the level to complete. To sum up, with bigger buffer sizes the granularity of task distribution is increased: workers receive fewer but heavier chunks of works leading to load imbalance.
3) **MPI Constraints:** Larger buffer increases communication latency and memory pressure, as each message involves more data movement and higher serialization and copying cost by MPI.

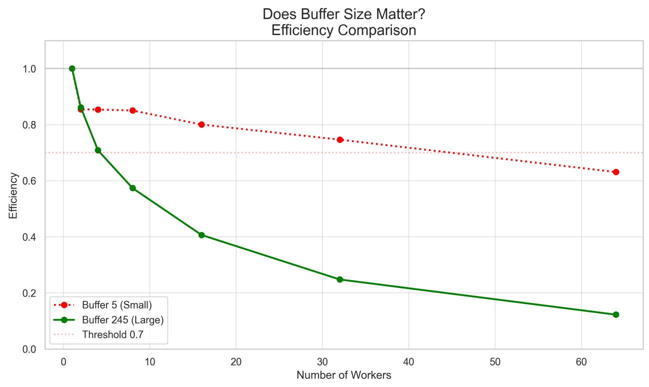

Fig. 2. Efficiency sensitivity to Buffer Size. The "Small Buffer" (Red line) significantly outperforms larger buffers at high worker counts, maintaining efficiency where larger buffers collapse due to load imbalance. Problem size used here is 1 000 000

Figure 2 further highlights this divergence. The efficiency of the Small Buffer configuration ($B = 5$) remains robust as workers increase, whereas the Large Buffer configuration ($B = 245$) sees efficiency plummet.

For this specific workload and cluster architecture, the latency of `MPI_Send` is negligible compared to the serialization cost of filling large buffers. Therefore, we identified the minimal buffer size ($B \approx 5$) as optimal, prioritizing *responsiveness* over *bandwidth saturation*, also given the fact that network capabilities of our cluster are very performative, as explained in the next section.

5. Benchmarking

Finally, we have evaluated performances of both algorithms on the UNITN HPC2 cluster. The cluster's calculation capacity is currently equals to 422,7 TFLOPs (theoretical peak performance CPU) and 7674 cores, allocated on 142 calculation nodes.

All nodes are interconnected with 10 Gb/s network. Some of them have an Infiniband connectivity (bandwidth increased up to 40 Gb/s) and some have an Omni Path connectivity (bandwidth increased up to 56 Gb/s). In any case, these specifications are more than enough for our case to ensure low-latency and high-speed connectivity for the MPI communication.

At the moment of performance evaluation, the operating system of the cluster was CentOS Linux version 7.9 and the scheduler used was openPBS version 19.1.

For MPI communication it has been used MPICH version 3.2 and the compilation tool used was gcc version 4.8.5.

To evaluate performances, we have implemented an appropriate `benchmarking.c` script for both algorithms: here we run the algorithm, with a given configuration of MPI processes and OpenMP threads (where each thread is executed on a single core), multiple times (5, both in the algorithm group implementation and in the paper implementation) and took the minimum execution time. We made this choice instead of the mean, because we wanted to take into account the best-case scenario of execution, avoiding that random noise or other processes running concurrently on the cluster would have damaged the performances.

Problem size explanation

In both algorithms. we considered as problem size the "computational complexity" of the integration function. To estimate it, we simply perform a certain number of "dummy" iterations to simulate a theoretical heavy functions. Therefore the number of iterations performed is our problem size.

5.1. First algorithm

To analyze the performances of our hybrid algorithm, we addressed benchmarking under two points of view: an OpenMP scalability (we fix to a single MPI process and we study the scalability of our application by increasing the number of threads) and a MPI scalability (we fix to a specific number of threads per process and we increase number of processes to analyze performances of the algorithm).

5.1.1. OpenMP scalability

Firstly, we wanted to analyze performances related solely with an OpenMP parallelization. This was used to find the optimal number of threads for each MPI process.

For job submissions, we asked always for 1 chunk with several cores (up to 16, to map each thread to a different core) and we required a *pack:excl* placement policy. This was done because we wanted to reserve the entire node of computation (where the chunk was allocated) entirely for us in order to avoid interferences from external processes and to collect more stable and reproducible measurements.

The problem size for this kind of experiments was set to 1 000 000.

TABLE II
OpenMP scalability results

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

Table II shows very positive results, suggesting that our algorithm could be very well parallelized using just a shared memory technique with up to 16 threads. In fact, the parallelization of the most computationally intensive parts with the functions' evaluations (3.2.3) is highly effective and dominates the overall execution time.

The scalability capabilities are very good since the algorithm is able to distribute the workload well to all the threads. As the number of threads increases, the runtime decreases almost linearly up to 16 threads, keeping an efficiency always safely above the acceptable threshold of 0.7. This indicates that parallel regions scale efficiently and overhead due to OpenMP directives remains limited.

Overall, the OpenMP scalability results show efficient shared-memory scalability, providing a strong basis for the hybrid approach. Based on these findings, we decided to assign **4 threads** per MPI process to accelerate serial sections while limiting threads overhead and enabling a broader MPI scalability analysis.

5.1.2. MPI scalability

After having validated the number of threads per process we started analyzing the scalability capabilities of our algorithm by increasing the number of MPI process. We fix 4 threads per process and started then evolve the number of process up to 64 processes.

We performed the benchmarking under two different PBS scheduling configuration. In the the first case, we enforced the *scatter:excl* placement policy. In the second case, this exclusivity constraint was removed. Under the exclusivity configuration, experiments were limited to a maximum of 32 processes, since allocating 64 MPI processes on distinct node with exclusive access to its resources would have resulted in excessive resource requests and prohibitively long queue waiting times.

As could be expected, the results obtained under the first configuration were much more stable across repeated runs and generally exhibited lower execution times. The table III shows the comparison. For these reasons, we adopted this configuration for the subsequent analysis.

TABLE III
Standard deviation results
(config 1: with *scatter:excl*, config 2: without)

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

a) Strong scalability

We analyze now the **strong scalability** properties with different sizes, all executed with the first PBS configuration described (apart from the case with 64 processes). To prove that the program is strongly scalable, it should maintain a constantly high efficiency without increasing the problem size.

TABLE IV
Strong scalability analysis

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

The table IV reports the results obtained by fixing the problem size and increasing the number of parallel processes. Overall, the algorithm exhibits good strong scaling behavior, particularly for medium to large problem sizes.

For all tested sizes, the speedup initially increases almost linearly as the number of process grows, indicating a good exploitation of the parallelism. A consistent deviation from ideal scaling starts to appear around 32 processes. In particular, the efficiency begins to noticeably decline when starting to use 32 processes and saturates or degrade badly at 64 processes.

This behavior is especially evident for smaller problem sizes (such as with 250 000 and 500 000 iterations): here the overhead associated with communication and synchronization becomes dominant already at 16 processes. As a result, the efficiency drops sharply and additional processes no longer provide meaningful speedup.

For larger problem size, the increased computational workload amortizes the parallel overhead and in that cases algorithm maintains **good efficiency up to 16** processes and **acceptable performances even with 32**. Nevertheless, even with heavy workload using 64 processes leads to an efficiency loss.

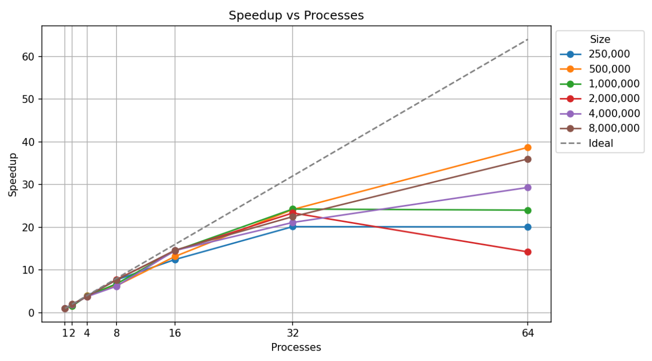

Fig. 3. Speedup evolution. This shows the near-linear behavior up to 16 processes and then a more flat one from 32 to 64.

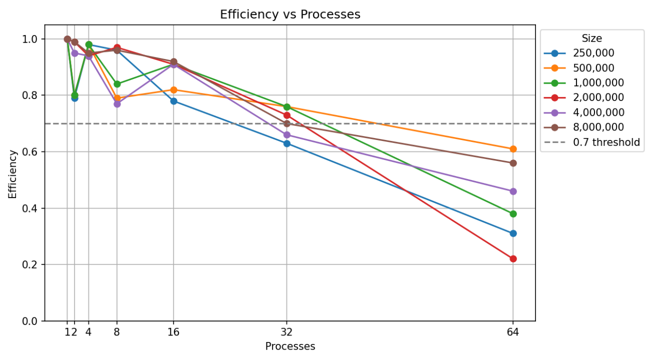

Fig. 4. Efficiency evolution. Efficiency always decreases after the peak around 4 process, but always stays above the decent threshold until 16-32 processes. No problem size permits to perform well with 64 processes.

b) Weak scalability

Last to analyze are the weak scalability properties. Ideally, we expect that if we double the problem size and the number of processes we manage to keep efficiency stable and quite high.

TABLE V
WEAK SCALABILITY ANALYSIS

Size	Processes	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

The results we got are very encouraging and indicate that the algorithm scales effectively under weak scalability conditions. Both execution times and efficiency remain remarkably stable as the problem size and the number of MPI processes increase proportionally. This behavior suggests that the algorithm successfully amortizes the overhead introduced by additional MPI processes when the computational workload grows accordingly. We can say that the cost of managing more processes is strongly outweighed by the improved workload distribution made possible by increasing parallelism.

Also here, as in the strong scalability analysis, we see an interesting deviation appearing at 32 processes: in that case, although the problem size is doubled, the efficiency begins to degrade noticeably. This suggest again the the algorithm approaches an intrinsic scalability limit, around that number of processes. Probably, at this scale communication overhead and synchronization costs start to dominate over the pure computation, reducing the benefits of a further parallelization.

Overall, this further analysis confirms that the algorithm performs **optimally approximately up to 16-32 processes**.

Finally, we would like to comment the small efficiency drop observed with two processes. We have attributed it to two main reasons. Firstly, with only two processes the ”fixed costs” (like MPI initialization and data distribution) represent a significant fraction of the total runtime and with just two processes these cannot be properly amortized over a larger amount of computation. Secondly, the problem size is still very limited and small compared to the later configurations. The workload per process is insufficient to fully ”hide” the communication and coordination overhead.

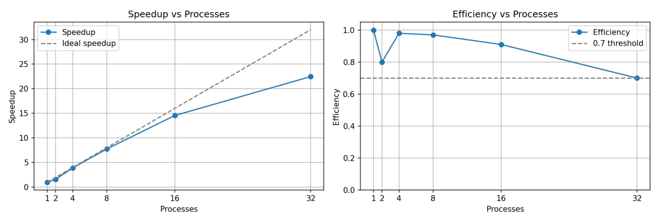

Fig. 5. Weak scalability evolution. We can see as efficiency remains more or less flat while increasing problem size and number of processes proportionally.

5.1.3. Conclusion

Overall results obtained by benchmarking are positives and allows us to find the parallelism limit of our algorithm.

From a shared-memory parallelization perspective, using **4 threads per processes** represents an effective compromise, balancing the number and the effectiveness of MPI processes with the OpenMP-level parallelization.

For all tested problem sizes the optimal configuration corresponds to **16 MPI processes**, while 32 processes still can provide decent results with larger problem sizes (from 1 000 000 iterations onward).

Consequently, the algorithm exhibits very good **weak scalability**, as performance remains high when both the problem size and the number of processes increase proportionally. In terms of strong scalability, performance degradation is more noticeable for smaller problem sizes, although results are (in any case) satisfactory even when a large number of parallel elements are involved (16/32 processes with 4 threads each).

5.2. Second algorithm (Master-Worker)

The benchmarking of the Master-Worker algorithm was conducted on the cluster environment with a standard PBS scheduling (without specifying any exclusivity constraint).

For strong scalability, we fixed the total computational complexity to 1 000 000 operations per point. The problem size was held constant while the number of workers increased. For weak scalability, we scaled the problem size linearly with the number of workers: $Complexity = 250,000 \times W$. This ensures that in an ideal scenario, the execution time remains constant as we add resources.

Crucially, in this Master-Worker paradigm, the Master process (rank 0) performs no integration work. Therefore, all scalability metrics are calculated based on the number of active workers ($W = processes - 1$), rather than the total number of MPI processes.

5.2.1. Strong scalability

We evaluated strong scalability by varying the number of workers $W \in \{1, 2, 4, 8, 16, 32, 64\}$ while using the optimal buffer size identified in our previous analysis ($B = 5$).

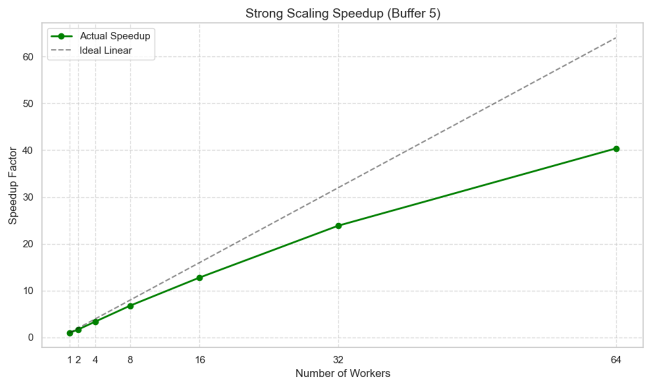

Fig. 6. Strong Scaling Speedup (Master-Worker). The algorithm achieves near-linear speedup up to 32 workers, with slight saturation at 64 workers due to Amdahl's Law.

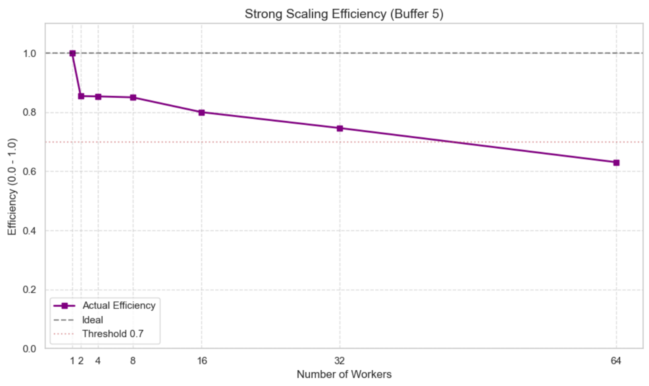

Fig. 7. Strong Scaling Efficiency. Efficiency remains remarkably high ($E > 0.8$) for most configurations, dropping only when the Master becomes a communication bottleneck.

Figure 6 shows the speedup factor. We observed near-linear speedup behavior. With low worker counts the execution time drops significantly, achieving a **3.41**× (from 111.63 seconds to 32.69 seconds) speedup when moving from 1 to 4 workers.

The system demonstrates strong scalability up to **32 workers**, where the efficiency is about 0.75. Beyond this point, although the speedup continues to increase (**40.4**× at 64 workers) the efficiency decreases due to the master becoming a bottleneck. This deviation from the ideal linear trend is expected: as the computation time per worker decreases drastically (sub-3 seconds), the serial overhead of the master managing the distribution loop becomes significant.

Figure 7 confirms this trend. The code maintains an efficiency $E \geq 0.7$ for all configurations up to 32 workers, demonstrating robust scalability.

5.2.2. Weak scalability

Weak scalability tests assessed the algorithm's ability to handle larger problems by adding more nodes.

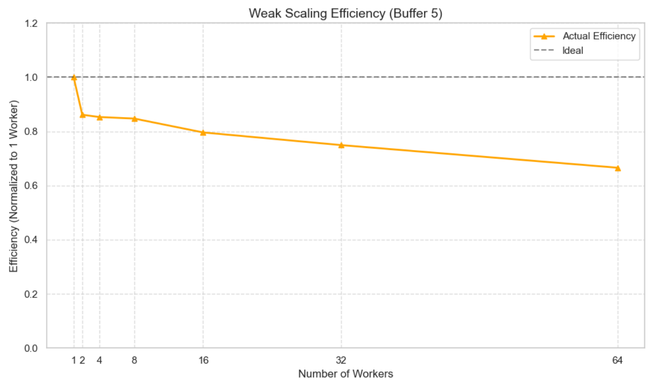

Fig. 8. Weak Scaling Efficiency (Normalized to $W = 1$). The stability of the curve indicates excellent weak scalability.

As shown in Figure 8, the efficiency remains close to ideal.

- At $W = 1$, the baseline time was 27.91s.
- At $W = 4$, with a problem size 4× larger (of 1 000 000 iterations), the time increased only slightly to 32.72s ($E = 0.85$).
- At $W = 64$, with a problem size of 16 000 000 iterations, the time was 41.92s, maintaining an efficiency of **0.67**.

The slight degradation at $W = 64$ is attributable to the increased communication volume the single Master must handle. However, the system successfully scales to solve a massive problem in nearly constant time, validating the design for large-scale cluster deployments.

The complete set of results is reported in Appendix F.

5.2.3. Buffer Size Impact

A unique parameter of this algorithm is the buffer size B. As detailed in Section 4.2, our benchmarks contradicted the reference paper's claim that B is negligible. We found that minimizing B (e.g., $B = 5$) yielded the best performance by reducing the "first-byte latency": the time workers wait for the Master to fill the first buffer.

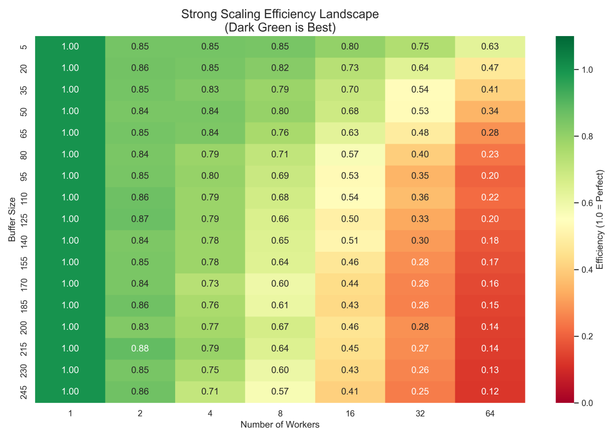

Fig. 9. Efficiency Heatmap (Workers vs. Buffer Size). Green regions indicate high efficiency. The heatmap clearly shows that smaller buffers (top rows) maintain high efficiency across all worker counts, while large buffers (bottom rows) cause performance to collapse at high W. For this set of experiments, problem size was set to 1 000 000

Figure 9 provides a comprehensive view of this phenomenon. The "Green Zone" (high efficiency) is strictly located in the upper region (Small Buffers). As we move down the Y-axis (increasing Buffer Size) and right on the X-axis (increasing Workers), efficiency degrades significantly (Red Zone), confirming that load imbalance becomes critical when large chunks are distributed among many workers.

5.2.4. Conclusion

Based on the experimental data, we can infer that the algorithm is **strongly scalable**. It achieves near-linear speedup up to **32 processors** ($E \approx 0.75$), while performance drops slightly at 64 workers ($E = 0.63$); this is a known limitation of centralized Master architectures on fine-grained tasks (sub-3 second duration).

From the weak scalability analysis, we saw that the execution time remains stable (increasing only by a factor of $1.5\times$) while the problem size scales by a factor of $64\times$. This proves the algorithm is suitable for solving massive integration problems on large clusters and is able to scale well when problem size and number of processors increase proportionally.

Finally, contrary to theoretical expectations, **small buffers (optimal at *B*=5)** are a critical parameter for performance on this architecture, maximizing worker responsiveness and minimizing Master serialization overhead.

6. Comparisons and final remarks

This paper showed two different parallel techniques to solve the same problem.

The two algorithms proposed show similar results: both are able to scale up well up to 16 processes, with the buffer-based algorithm that can perform well even with 32 MPI processes. The first algorithm is more sensitive to problem size: with bigger sizes it can easily scale up to 32 processes, while with smaller problem sizes it is difficult to achieve good efficiency results with more than 16 processes.

We consider both the techniques two well-designed scalable algorithms, as proven by our benchmarking.

A direct comparison with the results in [1] cannot be done, since in the original paper there are no indications about the computational complexity. In fact, the author did not specify what the size numbers were representing. Nevertheless, we can observe some interesting results.

From the experimental results in the paper, the buffer size was not an important parameter: it was not influencing benchmarking results at all. In our case, instead, it is crucial: in fact, with bigger buffer sizes, the efficiency of the algorithm starts to drop dramatically, starting from 8 workers allocated. In the paper scenario, maybe the experimental setup was different: it may relies on more computation-heavy kernel, where communication overhead is amortized.

From a performance point of view, we cannot compare directly the execution times but we can have a look at the speedup results (provided in appendix F). Paper algorithm shows poor strong scalability capabilities: in fact by doubling the number of processes involved every time, with small problem sizes performances degrading very quickly even with 8 and 16 processes involved. Nevertheless, with the biggest problem sizes analyzed performances are good, with a peak 0.88 of efficiency with 16 processes.

Overall our implementation seems to perform better, since it can reach positive results even with smaller sizes and we can increase more the number of processes. However, a proper comparison cannot be done without knowing precisely the metric used in the paper to measure the problem size.

In future works, further performance improvements of the master-worker algorithm could be explored. In particular, introducing OpenMP within the worker processes to parallelize the currently serial portions of function evaluations reducing execution time and improving overall efficiency.

References

[1] A. Yazici, "The romberg-like parallel numerical integration on a cluster system," 2009 24th International Symposium on Computer and Information Sciences, Guzelyurt, Northern Cyprus, 2009, pp. 686-691, doi: 10.1109/ISCIS.2009.5291906.

[2] Introduction to Numerical Integration, https://en.wikipedia.org/wiki/Numerical_integration

[3] Romberg's method explanation https://en.wikipedia.org/wiki/Romberg%27s_method

[4] Cluster specification https://servicedesk.unitn.it/sd/en/kb-article/cluster-hpc-architecture?id=unitrento_v2_kb_article&table=kb_knowledge&sysparm_article=KB0010434

[5] Cluster architecture specification https://servicedesk.unitn.it/sd/en/kb-article/differenze-tra-hpc2-e-hpc3?id=unitrento_v2_kb_article&sys_id=74f574de3be93210290f0854c3e45af6

Appendix A
General Romberg's method formula

Given an integral the $\int_a^b f(x)$ to approximate, we start with the step size $h = b - a$.

Initially, we must compute the trapezoidal rule halving the step size at each iteration until the desired number of sub-intervals on $[a, b]$ is achieved. This computes the *first column* of the Romberg table, denoted as $R[k][0]$:

- $R[0][0]$: approximation with 1 interval with width h
- $R[1][0]$: approximation with 2 intervals with width $\frac{h}{2}$
- $R[2][0]$: approximation with 4 intervals with width $\frac{h}{4}$
- and so on...

Secondly, we refine our approximation by combining the previous estimations using Richardson extrapolation formula:

$$R[n][m] = \frac{4^m R[n][m-1] - R[n-1][m-1]}{4^m - 1}$$

Basically the *m*th refinement of the trapezoidal rule with 2^n intervals depends on:

- the *m-1*th refinement of the trapezoidal rule with 2^n intervals
- the *m-1*th refinement of the trapezoidal rule with 2^{n-1} intervals

This allows the Romberg's method to be very adaptive, allowing the algorithm to stop the calculation as soon as the estimations difference is below a certain threshold, and to speed up the convergence, since it can reduce the error much faster than other Newton-Cotes formulas.

Appendix B
Points decomposition

The following picture shows the evolution of generation of sets E_n and I_n.
Blue points are the vertices, green points the edge nodes and red points the interior ones.

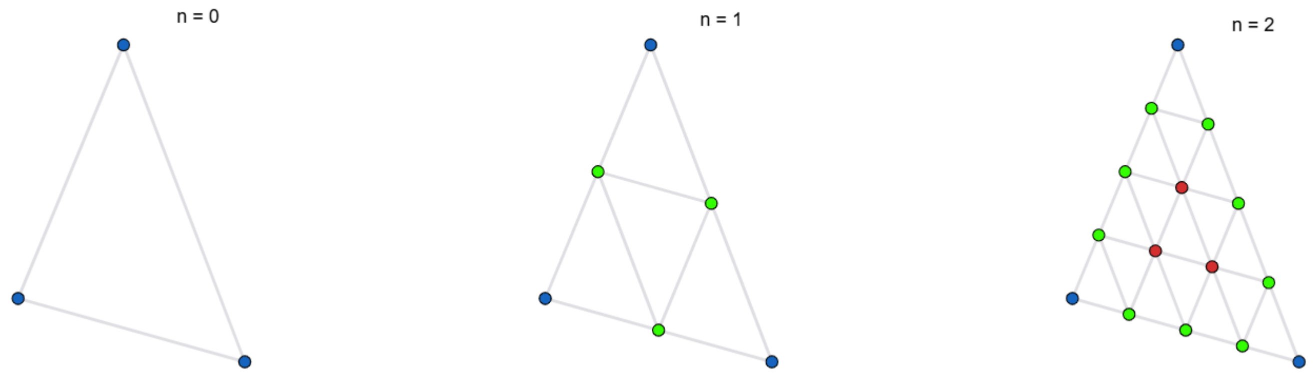

Fig. 10. Points generation

Appendix C
Data dependency analysis for algorithm 1

Here we show all the tables representing data dependencies of our algorithm.

C.1. Edge and interior points generation

TABLE VI
DATA DEPENDENCIES FOR EDGE POINTS GENERATION

Memory Location	Earlier Statement Line	Iter.	Access	Later Statement Line	Iter.	Access	Loop-carried?	Kind of dataflow	Issue
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)

```
1  int *edge_offsets = malloc(sizeof(int) * (nm));
2
3  edge_offsets[0] = 0;
4  for (int t = 1; t <= nm - 1; ++t) {
5      edge_offsets[t] = 3 * (t - 1);
6  }
7  ec = edge_offsets[nm - 1];
8  ec += 3;
9
10 #pragma omp parallel for
11 for (int t = 1; t <= nm - 1; ++t) {
12     double u = (double)t / nm;
13     int local_ec = edge_offsets[t];
14
15     Point pAB = lerp(A, B, u);
16     edge[local_ec++] = pAB;
17
18     Point pBC = lerp(B, C, u);
19     edge[local_ec++] = pBC;
20
21     Point pCA = lerp(C, A, u);
22     edge[local_ec++] = pCA;
23 }
24
25 free(edge_offsets);
```

Listing 1: Parallel code for edge points generation

TABLE VII
DATA DEPENDENCIES FOR 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)

Clarification on ”iteration i / $i+1$” in the double loop setting: we consider i to denote a single ”iteration” that takes into account the two loops indexes. So i represents an ”global” iteration (ia, ib) and $i+1$ another ”global” iteration where at least one of the two indexes differs from the previous one.

This means that these data dependencies occurs across any possible distinct pair of iterations.

```c
int *inter_offsets = malloc(nm * sizeof(int));
inter_offsets[0] = 0;
inter_offsets[1] = 0;

for (int indx = 2; indx <= nm - 2; indx++) {
    inter_offsets[indx] = inter_offsets[indx - 1]
    + (nm - 1 - (indx - 1));
}
int ic = inter_offsets[nm - 2] + 1;
int ib;
double u, v;
Point p;

#pragma omp parallel for private(ib, u, v, p)
for (int ia = 1; ia <= nm - 2; ++ia) {

    int ic_local = inter_offsets[ia];

    for (ib = 1; ib <= nm - 1 - ia; ++ib) {

        u = (double)ia / nm;
        v = (double)ib / nm;

        p = tri_interp(A, B, C, u, v);
        inter[ic_local++] = p;
    }
}
free(inter_offsets);
```

Listing 2: Parallel code for interior points generation

C.2. Distribution of points

```c

void compute_counts_and_displs(
int* sendcounts,
int* displs, int processes,
int total_length){

    //...

  int local_length = total_length / processes;
  if((total_length % processes) != 0){
    int indexOfProcess = processes-1;
    int left = total_length - (processes*local_length);
    for(; left > 0; left--){
      sendcounts[indexOfProcess] = 1;
      indexOfProcess = indexOfProcess - 1;
    }
  }

  //...

  displs[0] = 0;
  sendcounts[0] = local_length;
  for(int j = 1; j < processes; j++){
    sendcounts[j] += local_length;
    displs[j] = sendcounts[j-1] + displs[j-1];
  }
}
```

Listing 3: Serial code for points distribution

TABLE VIII
DATA DEPENDENCIES FOR POINTS DISTRIBUTION (1)

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)

TABLE IX
DATA DEPENDENCIES FOR POINTS DISTRIBUTION (2)

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**

```

void compute_counts_and_displs(
int* sendcounts,
int* displs, int processes,
int total_length){

    //...

    int local_length = total_length / processes;
    if ((total_length % processes) != 0) {
        int left = total_length - (processes * local_length);
        int t;

        #pragma omp parallel for
        for (t = left; t > 0; t--) {
            int indexOfProcess = processes - left + t - 1;
            sendcounts[indexOfProcess] = 1;
        }
    }

    //...

    displs[0] = 0;
    sendcounts[0] = local_length;

    #pragma omp parallel for
    for(j = 1; j < processes; j++){
        sendcounts[j] += local_length;
    }

    for (j = 1; j < processes; j++) {
        displs[j] = sendcounts[j-1] + displs[j-1];
    }
}
```

Listing 4: Parallel code for points distribution (not implemented in the project)

C.3. Local computations of partial sums

TABLE X
DATA DEPENDENCIES FOR PARTIAL SUM COMPUTATION

Memory	**Earlier Statement**			**Later Statement**			**Loop-**	**Kind of**	**Issue**
Location	**Line**	**Iter.**	**Access**	**Line**	**Iter.**	**Access**	**carried?**	**dataflow**	
local_sum	11	i	write	11	$i+1$	read	yes	FLOW	NO (use REDUCE clause)

C.4. Unparallelizable parts

TABLE XI
DATA DEPENDENCIES FOR RICHARDSON EXTRAPOLATION

Memory	**Earlier Statement**			**Later Statement**			**Loop-**	**Kind of**	**Issue**
Location	**Line**	**Iter.**	**Access**	**Line**	**Iter.**	**Access**	**carried?**	**dataflow**	
$R[m][k]$	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**

Note that if we would focus on parallelize the outer loop, as it is generally done when there are nested loops, the first data dependency would not cause problems, since the inner loop would have been executed sequentially.

Appendix D
Data dependency analysis for the Master-Worker algorithm

Here we show the tables representing data dependencies of the Master-Worker algorithm.

TABLE XII
DATA DEPENDENCIES FOR MASTER DISTRIBUTION LOOP

Memory	Earlier Statement			Later Statement			Loop-	Kind of	Issue
Location	**Line**	**Iter.**	**Access**	**Line**	**Iter.**	**Access**	**carried?**	**dataflow**	
`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**

TABLE XIII
DATA DEPENDENCIES FOR WORKER COMPUTATION LOOP

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

The complete data dependency analysis of the code, including the analysis of Richardson extrapolation part, can be found in the project's repository in the folder `paper_implementation`.

Appendix E
Results of the weak scalability analysis for the Master-Worker algorithm

TABLE XIV
WEAK SCALABILITY ANALYSIS (GROUPED BY BUFFER SIZE)

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
140	250 000	1	27.91	1.00	1.00
	500 000	2	33.80	1.65	0.83
	1 000 000	4	36.03	3.10	0.77
	2 000 000	8	43.26	5.16	0.65
	4 000 000	16	53.72	8.31	0.52
	8 000 000	32	93.96	9.51	0.30
	16 000 000	64	152.97	11.68	0.18
185	250 000	1	27.91	1.00	1.00
	500 000	2	32.30	1.73	0.86
	1 000 000	4	37.36	2.99	0.75
	2 000 000	8	45.88	4.87	0.61
	4 000 000	16	64.91	6.88	0.43
	8 000 000	32	110.96	8.05	0.25
	16 000 000	64	195.20	9.15	0.14

Appendix F
Results of the implementation in [1]

To know more about system setup see [1]. We considered the benchmark done with buffer size 100 bytes (that leads to 6 points capacity, comparable to the buffer size of 5 points used in our implementation).

Here p was considered as the total number of processes and not the total number of workers.

TABLE XV
PERFORMANCE FROM THE PAPER ANALYZED

Size	p	2	4	8	16
10	S	1.00	2.41	4.79	4.24
	E	0.50	0.60	0.60	0.27
20	S	1.00	2.66	5.66	8.15
	E	0.50	0.67	0.71	0.51
40	S	1.00	2.81	6.48	14.07
	E	0.50	0.70	0.81	0.88
80	S	1.00	2.92	6.59	13.72
	E	0.50	0.73	0.82	0.86