A Finite Volume Case Study From An Industrial Application

Miguel Palhas, *pg19808, MEI* , Pedro Costa, *pg19830, MEI* and Stéphane Clain, *co-Advisor*

Abstract—This report presents the analysis, optimization and parallelization of `polu`, an application which computes the spread of a material in a bidimensional mesh. Three parallel approaches were designed: shared memory using OpenMP; distributed memory using MPI; and a GPU version using CUDA. All the implementations obtained speedups, but the scalability was found to be only relevant with OpenMP and CUDA. Despite the efforts to optimize it, all the implementations presented locality problems. The best results were achieved with the CUDA implementation. Better speedups are expected with a larger test case, but such could not be generated for this project.

Index Terms—Integrated Project, Finite Volume, mesh, C/C++, OpenMP, MPI, CUDA

1 Introduction

This document describes an incremental work where the `polu` application is studied with the goal of improving its performance. The program uses a Finite Volume method to simulate the spread of a material in a 2D surface.

The goal of this project is to study different approaches to parallelize `polu`. Three different parallel implementations are used: the first using shared memory; the second using distributed memory; and a third implementation using a GPU for a massively parallel approach.

A total of four different stages of development contributed for the results shown in this document. In the first stage, the original implementation was deeply optimized and analyzed to allow parallelism, and two versions were implemented using shared memory and CUDA. The second stage evolved the shared memory version by further optimizing the sequential version and parallelizing it. A third stage focused in developing a naive but functional distributed memory implementation, with almost no optimizations. Based on the results of the previous stages, the final stage was set to focus in improving the initial CUDA version.

In this final document, the previous stages are summarized according to the implementations developed. Section 2 describes the case study and the analysis which allowed to identify the parallelism opportunities in the algorithm. The sequential versions implemented, from the original code to the best optimization achieved are discussed in section 3. Sections 4 to 6 describe the approach, load balance and limitations of the shared memory, distributed memory and massively parallel implementations, respectively. Final comparative results are shown in section 7 and a project conclusion is presented in section 8.

Several details, due to the non-essential nature of the subjects, were pushed to appendices A to C. These do not include details such as intermediary results obtained in previous stages of the project, which are omitted from this document. Only final speedup values are shown. Please refer to the previous reports for further details about intermediary stages.

- *M. Palhas and P. Costa are with the Department of Informatics at University of Minho, Braga, Portugal*
  E-mail: pg{19808,19830}@alunos.uminho.pt
- *Professor Doctor Stéphane Clain is with the Department of Mathematics and Applications at University of Minho, Braga, Portugal*
  E-mail: clain@math.uminho.pt

2 Case Study

The application analyzed in this document, here called `polu`, computes the spread of a material (e.g. a pollutant) in a bidimensional surface through the course of time. This surface is discretely represented as a mesh, composed mainly of edges and cells. The input is given in the form of XML files, passed as arguments:

- Mesh description;
- Velocity vector for each cell;
- Initial pollution values of each cell.

Both the input and the output of the program can be converted to the `msh` format, compatible with the `gmsh` mesh generator, for data visualization. This is done using conversion programs, also written using the FVL library, which handle the conversion between the XML schema used by the library, and the mesh format specification of `gmsh` [1].

2.1 Algorithm

The algorithm used by this application is a first order finite volume method. This means that each mesh element only communicates directly with its first level neighbors in the mesh, which makes this a typical case of a stencil computation [2]. In terms of performance, being a stencil algorithm implies that the operational intensity will most likely remain constant with larger problem sizes [3], [4]. On the other hand, the low order allows for a greater locality of the calculations, and favors parallelization.

The code consists on a preparation stage, where all the required elements are loaded and prepared, and two computation stages, which compose the main loop.

Operations performed in the preparation stage are highly dependent on the implementation being described, as most will require some elements to be properly organized or some values to be previously computed. Common operations, such as loading the necessary data from the described files are constant to every implementation, but may still differ in the structures used to store the data.

A single execution of the two computation stages together form a step in the iterative method behind this application.

These stages, also referred in this document as core functions, or kernels, are the `compute_flux` and `update` functions.

In `compute_flux`, all the edges in the mesh are analyzed, and the flux of pollution to be transfered across that edge is computed, based on the pollution level and the velocity vectors of the adjacent cells. A preconfigured value is used as the *Dirichlet* condition[1], which replaces the pollution level of a second cell for the edges in the border of the mesh.

As for the `update` function, it uses the computed flux values to update the pollution levels of each cell in the mesh, by adding the individual contribution of each edge of the cell. While triangular cells are preferred, there are no restrictions to the number of edges a cell may have.

2.2 Parallelism Opportunities

A typical approach to parallelism in stencil algorithms is based on the premise that a given point of the domain can be computed once the surrounding points (its dependencies) are ready. Due to this being a first order algorithm, these dependencies are extremely local (computations for a cell depend only on the immediately adjacent edges, and vice-versa).

Theoretically, this should the basis for the parallelism strategy: mesh subsets can be computed once their directly connected elements are up-to-date, even though other unrelated mesh subsets would possibly be in an older state. However, such an approach also adds a higher degree of complexity, and involves deep changes to the underlying algorithm. As the purpose of this document is not to study a better alternative algorithm to solve the problem `polu` is aimed at, this approach was not followed.

Instead of the typical strategy, each core function can be analyzed as an independent task. During `polu`'s main loop both `compute_flux` and `update` depend on each other to perform their tasks. `update` requires flux from all edges to be previously computed in `compute_flux`, which in turn requires that all pollution values are up-to-date to correctly compute the flux for the next iteration. This creates two implicit synchronization points in the main loop and is a consequence of the heartbeat characteristics of the problem. Yet, both functions may be completely executed in parallel, performing the required computations for each internally iterated mesh element.

3 Sequential

In this section are presented the sequential implementations of `polu`, from the original to the optimized versions. Section 3.3 analyzes the dependencies in the sequential code which prevent the parallelization of the core functions (as explained in section 2.2), and how the code was adapted to remove them.

3.1 Original

The original `polu` program was implemented using structures as *Arrays-Of-Pointers* (AOP). While this allows for an easier development and better code readability, the deep levels of dereferencing imply an increased number of memory accesses, which aggravate the effects of any memory bottlenecks.

1. The *Dirichlet* condition is a type of boundary condition used to specify a value taken by the solution in the border of the domain. In the `polu` application, this value is constant for the entire mesh border and throughout the execution.

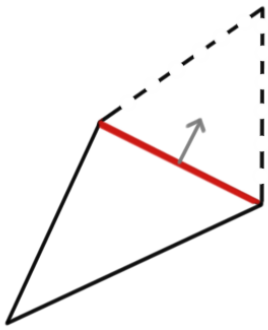


Fig. 1: An edge. Every cell has at least one adjacent "left" cell. Border edges do not have a "right" cell. Edge normals always point to the "right".

In this original version, the `compute_flux` function (see fig. 2) iterates over each edge, and loads the pollution levels and velocity vectors of the cells it connects. A convention is followed (see fig. 1), indicating that an edge always has a cell on its "left", and that the normal of the edge always points to its "right" cell. For the edges in the border of the mesh, the "right" cell does not exist. In this case, the *Dirichlet* condition is used as a the polution value in that hypothetical cell.

The value for the velocity in the edge is computed based on the velocity vectors of both adjacent cells and the normal of the edge. The edges in the border assume for the hypothetical "right" cell a velocity equal to the "left" cell, which results in null flux. Despite not being a vector, this computed velocity gives information about the direction of the flux: a positive value represents the pollution flow from left to right; the inverse happens with a negative velocity.

This function also returns the elapsed time, which is computed with the maximum absolute value of the computed edge velocities, and is used at the end of the iteration to keep track of the total elapsed time.

---

for all $edge \in Edges$ **do**
  $l \leftarrow left\_cells_{edge}$
  $r \leftarrow rigth\_cells_{edge}$
  $u_l \leftarrow pollution_l$
  if $\exists Cells_r$ **then**
    $u_r \leftarrow pollution_r$
  else
    $u_r \leftarrow Dirichlet$
  end if
  $v_{edge} \leftarrow \frac{\vec{v}_l + \vec{v}_r}{2} \cdot \vec{n}_{edge}$
  $v_{max} \leftarrow \max(v_{max}, v_{edge})$
  $flux_{edge} += u_l \cdot [v_{edge}]^+ + u_r \cdot [v_{edge}]^-$
end for
$\Delta t \leftarrow |v_{max}|^{-1}$
return Δt

Fig. 2: Pseudocode for the original `compute_flux` function.

The original `update` function also iterates over each edge. The contribution of each edge to the cells final value is computed as the product between the elapsed time, the computed flux and the ratio between the edge's length and the cell's area. This is illustrated in fig. 3.

In this original implementation, there is also the possibility

```
for all edge ∈ Edges do
    Δu ← Δt · flux_edge · L_edge
    l ← left_cells_edge
    r ← right_cells_edge
    pollution_l −= Δu / A_l
    if ∃Cells_r then
        pollution_r ← Δu / A_r
    end if
end for
```

Fig. 3: Pseudocode for the original `update` function.

to output the current state of the mesh after every X iterations, where X is a parameter provided in the configuration file loaded at runtime. This feature allows the output to contain not only the final state of the system but all the intermediary states, allowing an animation to be shown using `gmsh`.

3.1.1 Simplifications

Two important simplifications were performed in the original version, before any other adaptation or rewrite of the code.

The first simplification was the suppression of the output operation at the end of each main loop iteration, thus removing the animation feature. While this feature is interesting to analyze how the system evolves during the simulation, input/output operations are slow and dependent on system calls. Since the main goal of this document is to study ways to improve the performance of the `polu` application, this feature would introduce a large overhead, preventing possible optimizable points from being identified correctly.

The second major simplification focused on the `compute_flux` function. Since the velocity vector of every cell remains constant throughout the entire program, the same values will be computed for the velocity in each edge, and, consequently, for the elapsed time. While this computation is important in a more dynamic application where velocity vectors are not constant, such is not the case for this algorithm, as no function is implemented to change these vectors. Therefore, it is possible to remove these two computational steps ($v_{\max}$ and Δt) from the core function to the preprocessing stage, improving global performance.

3.2 Optimizations

While optimizing the sequential code may be important to eliminate possible bottlenecks created by inefficient memory access patterns or poorly written code, most of the optimization work was kept for after the parallelization, as early optimizations might have hampered parallel implementations. As such, the focus of optimization in the sequential implementation of `polu` was on the structures used, which were trivially found to be very inefficient.

As previously stated, the original implementation relied on an AOP approach, but the excessively deep chains of pointers translate into more memory accesses, and consecutively worse performance.

The next two sections describe the two alternative approaches and explain how these improved the application's performance.

3.2.1 AOS

The first implemented alternative to AOP was the *Arrays-Of-Structs* approach. Instead of using pointers, structures were created for cells and edges, containing all the data related to each. Where pointers previously existed to link an edge to the adjacent cells, or a cell to its edges, an index is placed. While the dereferencing levels are completely eliminated, using indexes provides any element with identifiers which allows direct access to the other elements it needs to interact with, as long as all the data objects are stored in arrays. The maximum representable index is used as the equivalent to `NULL` pointers (applied, for example, as the identifier for the "right" cell of a border edge).

3.2.2 SOA

While the AOS approach completely removes the need for dereference, using a *Struct-Of-Arrays* (SOA) approach can be more efficient.

One of the problems with the AOS approach is that when the core functions iterate over one of the arrays, cache lines are being filled with the complete structures of these elements. Yet, neither of the core functions utilizes all the data in that structure, which translates in useless data occupying the cache. This hurts spatial locality[2], greatly increasing the chances of accessing a mesh element translating into a RAM access.

The SOA approach solves this problem by placing each piece of the elements data in a distinct array. As an example, a single array is created to hold the pollution level of all the cells. The same applies, for example, with the flux of all the edges.

This approach allows the core functions to load only the data required for their computations. The cache lines are now filled only with useful data, and the pieces which are required by the other functions will only be loaded when necessary.

3.2.3 Results

To evaluate the impact of the optimizations presented in sections 3.2.1 and 3.2.2, performance tests were performed with the original and optimized versions. Appendices A and B describe the environmental setup and methodology for the tests performed, respectively.

Figure 4 shows the results obtained for the three sequential versions. Both optimized versions show a clear improvement just by not using the several dereference levels. Also, the results show a clear improvement by using SOA over AOS, proving the effectiveness of the optimization.

3.3 Limitations

The main limitation for the sequential versions is the locality of the mesh used in the problem. Since this issue will be present in any shared memory implementation too, that discussion is left for section 4.2.

Figure 13 shows the roofine for SeARCH Group 201 with the computational intensities of the SOA and AOS versions. Computational intensity is defined to be the ratio between all the instructions not related with memory accesses (loads and stores) and the number of bytes accessed in RAM.

The improvement from AOS to SOA can be explained only by the improvements in locality. Further testing, which results were omitted in this document, shown that the number of

2. If a data element is required, adjacent elements will likely be required in the near future.

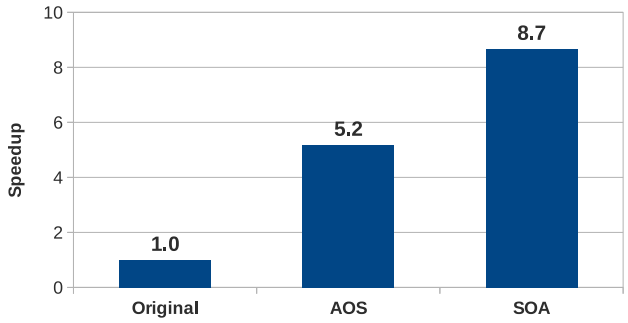


Fig. 4: Speedup results for the three sequential versions, compared with the original version.

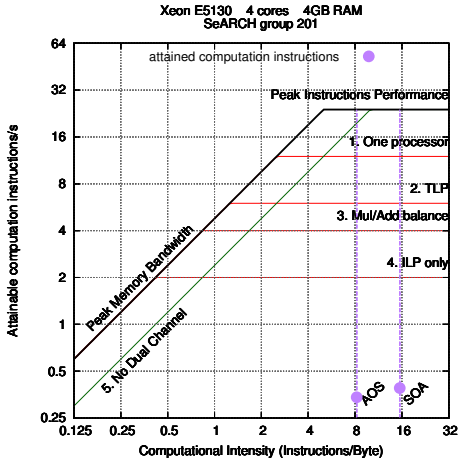


Fig. 5: Roofline for SeARCH Group 201, with the computational intensities of the AOS and SOA versions.

bytes accessed to RAM, number of computational instructions completed and the core functions execution time all decreased, achieving almost the same amount of computational instructions per second, while halving the number of accesses.

3.4 Dependencies

As explained in section 2.2, the heartbeat nature of the algorithm prevents extending parallelization beyond the scope of each core function, but both functions are able to execute their internal iterations in parallel. Yet, some dependencies exist among the elements used in these functions, and adaptations are required to allow parallelization.

`compute_flux` has only one dependency, in the computation of the elapsed time Δt, which is removed simply by accepting the second simplification described in section 3.1.1. While this simplification is assumed in the final implementations, this dependency was properly studied in early stages of the project. Since the computation of Δt is based in a maximum operation, it is possible to replace this dependency with a reduction after the computation of the fluxes, thus removing the dependency and maintaining the correctness of the program.

On the other hand, the `update` function has a dependency which may not be neglected because of a simplification. As described in section 3.1, the original implementation of `update` iterates over the edges. While this may favor locality after executing `compute_flux`, it creates a race condition when each edges adds its own contribution to the cell's value. This dependency can be removed by changing how `update` iterates over the mesh. By iterating over the cells instead of the edges, the race condition disappears since only one write operation per cell is performed, as shown in fig. 3.

---

for all $cell \in Cells$ **do**
  for all $edge \in Edges_{cell}$ **do**
    $pollution_{cell} += \Delta t \times flux_{edge} \times \frac{L_{edge}}{A_{cell}}$
  end for
end for

Fig. 6: New `update` function, now iterating over cells instead of edges

4 Shared Memory

A shared memory parallel version of `polu` was implemented using the OpenMP interface. The main feature about this API is that the program itself remains almost equal to the sequential version, requiring only the addition of the necessary directives and the adaptation of the code to remove dependencies.

Both core functions were parallelized just by adding a `parallel for` directive to the internal loops. The number of threads issued in each parallel zone is given as the second argument of the program, and defaults to the maximum number of threads supported by the hardware in the absence of the argument.

4.1 Load Balance

Both core functions are mostly homogeneous in its parallel implementation. In `compute_flux`, the two branches perform the same amount of operations whether they are followed or not. In `update`, the heaviest part of the workload is constant (products and divisions), and it only differs in the number of edges contributing to the cell. While this value may cause the function to become heterogeneous, this is highly dependent on the mesh used (the test case used in this document has 3 edges in every cell).

By default, the OpenMP interface uses static scheduling, where iterations are assigned to threads in a *round-robin* pattern. Since the number of edges and cells is a constant throughout the entire execution, this guarantees the best load balancing among threads.

4.2 Limitations

The main problem with this implementation is data locality. While these issues are softened using a SOA approach, and the method is a first order one, the algorithm is still not very cache friendly in either of the core functions.

In `compute_flux`, each edge requires access to data from the two adjacent cells. While the access to edges is contiguous here, access to cells is not for most of the iterations. Yet, border edges do not require the second edge.

`update` on the other hand requires access to all the edges in a cell, which are always more than two. Analogously, the access to cells here is contiguous, but the access to edges is not. Since there are more edges per cell than there are cells per edge, this function is most likely the bottleneck of the main loop.

The locality issues may be improved by changing how the mesh is built. The generator used for this project (`gmsh`) does not optimize the mesh locality (see fig. 7).

Alternatively, works from other authors were found to improve this issue (the most promising being [5]), but the complexity behind studying a new approach to the problem and implementing it could push the project beyond the time limitations.

Similarly, a specialized library to work with meshes was found [6]. While it could dramatically improve the mesh structure, just like using other authors' works, setting up the library and learning how to use it properly could take more time beyond the limits of the project.

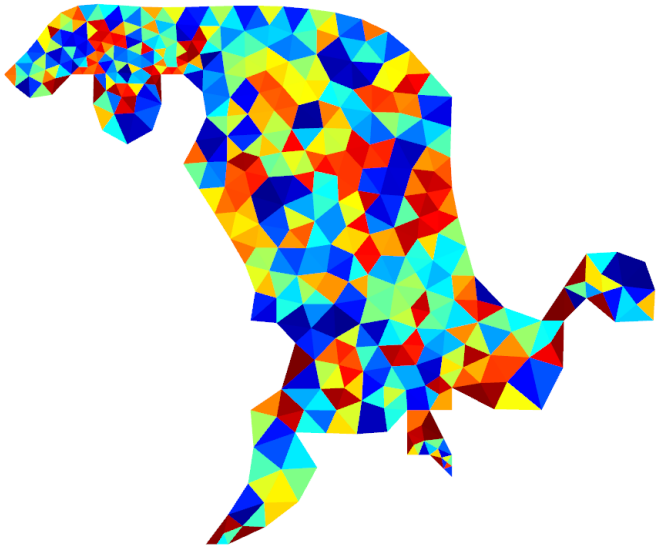


Fig. 7: Cell locality in the mesh used as test case in this document (with increased granularity, to make it visible). The more different the colors, the more distant the cells are in the data structure.

4.3 Results

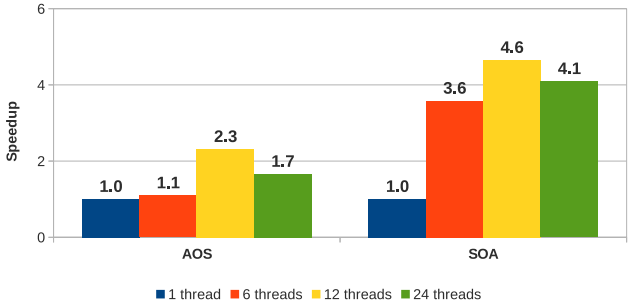
Figure 8 shows the speedups obtained with the two optimized sequential versions parallelized using OpenMP, for different numbers of threads.

It is clear to see that the SOA version obtained the best results. Even considering the speedup obtained compared to the sequential AOS version, this approach to the structures implementation scales better (fig. 8a), allowing to take further advantage of the hardware resources.

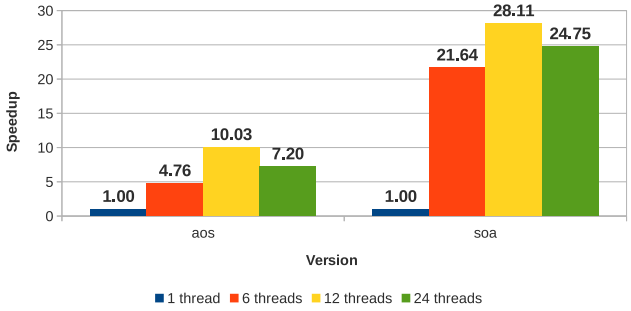
The best results were obtained using 12 threads, which in SeARCH Group Hex nodes represents the full hardware support, disregarding Intel® Hyper-threading technology. While this allows two threads to run in the same processor core, it rarely is beneficial for the program execution.

5 Distributed Memory

For a distributed memory implementation, the *Message Passing Interface* (MPI) was used. With the sequential code having already suffered some changes and optimizations, the main problem consisted in the partitioning of the mesh. The obvious approach is one where each processor is assigned to a subset of the entire mesh, and is responsible for the application of both kernels to its subset. Communication is also required between each main loop iteration, since each process will require access to the values in the border of its subset, in order to compute its own values.



(a) Compared with the respective sequential implementation.



(b) Compared with the original implementation

Fig. 8: Speedup results for the two optimized sequential versions, parallelized with OpenMP.

5.1 Mesh Partitioning

In order to distribute processing payload across each process, the input mesh, and all of the data associated with it, needs to be split into partitions. A mesh partitioning algorithm is required. This algorithm must generate P disjointed partitions (where P is the number of processes), each to be assigned to a different process. Some additional data is also required for each partition, so that information about how each partition connects to the others is kept, to allow communication to be done correctly.

5.1.1 *Research in Mesh Partitioning*

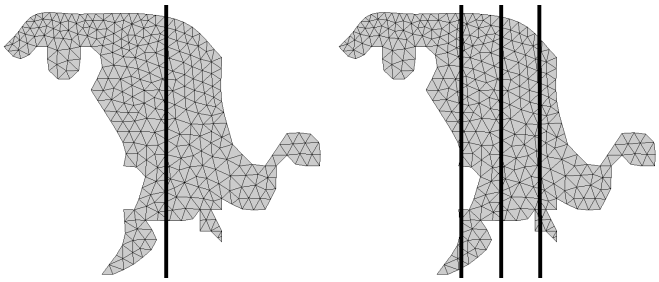
Mesh partitioning is currently an actively researched topic, with some projects and libraries being already available to help understanding how a mesh (or more generically, a graph) can be partitioned in ways to optimize certain aspects like load balancing, communication balancing or partitioning overhead. For instance, in [6], a library called `Metis` is presented whose purpose is precisely this problem. Other works found for this topic include [7], [8].

However, given the already mentioned time constraints of this project, and since the priority was to have a functional algorithm rather than an optimal one, it was decided not to attempt an approach involving `Metis` or any other researched work.

5.1.2 *Partitioning Methodology*

The algorithm created to partition the mesh works by dividing it in slices by the horizontal coordinate of each cell. Given

a mesh with a total of C cells, and a pool of P processes, the mesh is divided into P partitions, each with exactly $N = C/P$ cells, differing by at most one cell when they cannot be evenly divided. By ordering the cells based on their horizontal coordinate, they are sequentially assigned to each process, in such a way that the first process receives the first N cells of the set, and so on.



(a) 2 partitions (b) 4 partitions

Fig. 9: Mesh partitioning illustration

With this partitioning method, communication is also greatly simplified, as it is guaranteed that every partition will have a left and a right neighbor. It is also assumed that the width of the global mesh is big enough so that there are never processes with no cells assigned, which would break communication. Since a common mesh usually contains at least thousands of cells, this should not be an issue. The partitioning and assignment of the cells is done sequentially, thus it will increase preparation time.

5.2 Communication Analysis

When computing the flux for a given edge, the pollution values of both adjacent cells are required. Until now, only one divergent case existed, when the edge was in the border of the mesh. With the addition of partitioning, a new divergence is created, when the edge is not in the border of the global mesh, but in the border of the local partition, meaning that one of its adjacent cells was assigned to a different partition.

A communication step is required at this point, so that each edge in the local border (the border that connects to another partition, not the global mesh border) receives the corresponding values from the neighbor partition

This communication step was introduced at the beginning of the main loop, and consists of two smaller steps, one for left communication, and one for right communication. In each iteration, every process starts by communicating the left border values, which were previously indexed in the preparation stage, to its left neighbor. Asynchronously with that task, it receives an equivalent message from the right neighbor, which is also at the same step. After both tasks, the direction of communication is reversed, and the right border values are sent to the right neighbor of each partition.

Only after all communication is done for this process is it able to proceed to the main kernels of the loop. This introduces a large overhead, as it will most likely require a network transfer if the neighbor partition is located in a different machine. This overhead may become a huge bottleneck for the loop, especially for smaller inputs, where the time spent in the kernels is small enough to make the partitioning and communication occupy a large percentage of the program.

An alternative could consist in dividing the communication into two asynchronous tasks, allowing the flux for inner edges to be computed while the communication is taking place, since they don't depend on the values to be received. Again, due to time constraints, it was not possible to analyze this solution further.

5.3 Load Balance

With the naive partitioning strategy used, load balance becomes a problem. The computation itself is actually well balanced, since it is assured that every partition has the same amount of cells (differing at most by one). The problem is in the border between those partitions. With the division by the horizontal coordinate being used, it becomes obvious that the size of the border between partitions becomes extremely dependent on the format of the mesh itself. Other approaches, already mentioned in section 5.1.1 attempt to deal with this, and produce partitions that minimize the size of the border.

The drawback of not controlling border sizes comes at the communication step. Not only a different partitioning solution could minimize the border, thus minimizing the amount of data transfered, it can also happen that different partitions have very different border sizes, compromising communication balance.

5.4 Results

These results show the attained speedups for the MPI implementation. Due to SeARCH limitations, only tests with two nodes were able to run (see appendices A and B for further details regarding the environmental setup and methodologies used, respectively). Speedups are compared against the original sequential version in fig. 10.

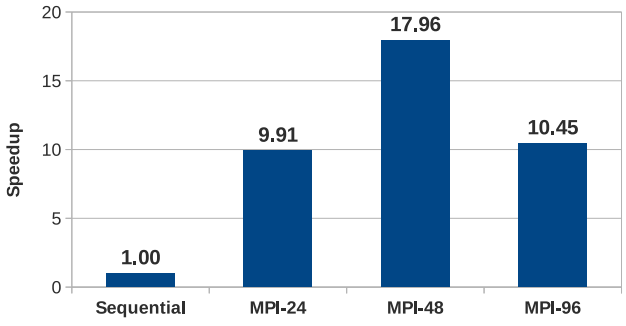


Fig. 10: MPI implementation speedups.

While there are actual speedups, especially using 48 processes, the results are not as good as the shared memory results. The reason for this comes from the nature of the algorithm, that makes it very sensible to the overhead of communications. This imposes large limits to the scalability of any distributed memory implementation of `polu`.

6 GPU

The development of a GPU implementation of `polu` was divided into two phases. The initial implementation consisted only on getting a working implementation, with basic, somewhat naive kernels implemented to move `compute_flux` and `update` computations to the GPU. After a comparison with the other implementation done at that point, it was decided to invest more time in the CUDA version. Not only was its

execution time much better than other implementations, since it was still a naive implementation it was most likely to still have room for improvements.

6.1 Load Balance

The nature of the algorithm allows for an easy and effective load balancing strategy. In CPU implementations, `compute_flux` iterates over all edges, and `update` over all cells, and the workload of each iteration is homogeneous and free of dependencies, as already shown in section 4.1. So the balancing strategy can consist, for both kernels, in simply assigning one iteration to one CUDA thread. In `compute_flux`, one CUDA thread will be responsible for computing the flux for a specific edge, while in `update`, one CUDA thread will update the pollution of a specific cell.

In order to use the ideal block size for each kernel, the CUDA Occupancy Calculator [9] was used, having in consideration the CUDA Capability of the device used (see appendix A) and the amount of registers and memory used by each kernel, which is accessible via nvcc. Using the block size recommended by the CUDA Occupancy Calculator does not guarantee better performance, and may actually result in unwanted behavior, such as register spilling, or greater branch divergent. Experiments were made with different block sizes to detect which provided better results.

6.2 Optimizations

One of the first optimizations performed comes from a simplification already explained in section 3.2. The initial implementation, like the CPU versions, relied on a reduction to compute the maximum velocity after each iteration. The reduction used was based on the most optimized implementation provided in the NVidia samples, and is, according to the author, one of the most optimized CUDA reductions.

But the simplification of moving the maximum velocity computation to preprocessing stage did not rely only on the reduction, as that was only the final step of the computation. There was also added workload to the `compute_flux` kernel, which had to compute the velocity for each edge. More than the amount of operations, this had a considerable impact on the amount of registers used by the kernel, allowing for more threads to be spawned without having register spilling, and it was noted that after its removal.

6.2.1 Second Phase

On the second phase of the development of a CUDA implementation, more attention was given to the individual performance of the kernels. Several experiments were performed with different kernel implementations, in order to test different approaches to eliminate memory accesses and reduce or eliminate divergent branches.

While this provided small improvements to kernel execution time (consequently decreasing the main loop time considerably) the better speedup was achieved after a division operation was removed from `update`, which computed a ratio between the length of an edge and the area of a cell. This was achieved by precomputing a matrix with the value of the ratio in the preprocessing stage. This required a considerable amount of additional memory to be used on the GPU, but since the problem is not bound by total size (largest test case used consisted of less than 100MB) and division operations are very costly to a GPU, this showed large improvements for the `update` kernel.

	Initial	Optimal	Speedup
`compute_flux`	0.72	0.7	**1.03**
`update`	3.60	1.96	**1.84**

TABLE 1: Comparison of first and last kernel implementations. Times are in ms.

Times shown in table 1 were measured using the CUDA Events API, which allows measurements of GPU events, excluding the callback overhead of issuing a kernel call. That, along with the fact that both kernels have considerably short and simple code, explains the low times, even for the initial versions. Even so, the removal of the division operation, along with other smaller tweaks through branchless operations, allowed the execution time of the `update` kernel to be almost halved.

Due to the low precision of the measurements when compared to the small execution time of the kernels[3], no immediate changes can be seen between some of the versions. However, since later kernels are the result of aggregating most optimizations from previous versions, intermediate results are still influent.

6.3 Mesh ordering impact

Since GPUs rely heavily on memory organization, it comes as no surprise that the issues stated at section 4.2, and shown in fig. 7 become even more relevant.

Memory accesses in a GPU are more efficient when the data being accessed at the same time is contiguous in memory, allowing for a single, coalesced memory request to be issued. The lack of locality provided by `gmsh` output, as well as the irregular memory access pattern from each kernel complicates this task.

A simple approach was taken to attempt minimizing this problem, which consisted on preprocessing the mesh, moving all cells in the border to the beginning of the structure. The idea was not to increase locality, since the border is only a small subset of the entire mesh, but to attempt more regular accesses in `compute_flux`, as well as avoiding the divergent branch for border cells. With all border cells being sequential, then only a single warp should diverge on the branch which tests borders.

This solution showed no visible improvements, probably due to the already extremely low kernel execution times, which become bottlenecked by the kernel callback overhead.

6.4 Results

Results for the CUDA implementations are here presented in the form of speedups, relatively to the original raw version of `polu` (see appendices A and B for details on the environmental setup and methodology used, respectively). Results were measured by comparing both the first, more naive CUDA implementation, and the last implementation, with better kernels and the division removed from `update` kernel. Some intermediate implementations were also created, since optimizations were aggregated by each new kernel version created, but only the final results of the optimizations are shown in fig. 11.

The initial CUDA version already showed significant performance gains, almost matching the best speedup achieved

3. CUDA Events API claims a precision of only 0.5 ms

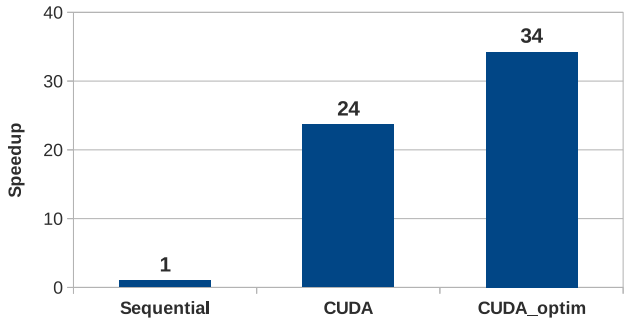


Fig. 11: CUDA implementation speedups

with the OpenMP implementation in section 4. Optimizations were able to boost this speedup even further, achieving a total speedup of 34, the best one in this project.

However, the gain against the OpenMP version is not as large as previously expected, since this problem seemed extremely well suited to a massively parallel approach, like the one employed by CUDA. This is a result of the limited test case size, which was generated using a conversion utility provided with `polu`, that converts `gmsh` output to a XML file readable by `polu`. This utility was not subject to optimizations or parallelization, but uses an algorithm in the order of $\Theta(N^3)$, making the generation of larger test cases impossible due to time limitations. The test cases available, which only go up to 62MB, still do not reach the maximum capacity of the GPU, which may benefit from the addition of even more threads.

Given the chance to test this theory, this could show much greater scalability for the CUDA implementation.

7 Final results

Since the individual results for each implementation were shown in the respective section, this section shows the global comparative final results.

Figure 12 shows the final speedups obtained for best version of each implementation, compared to the original implementation.

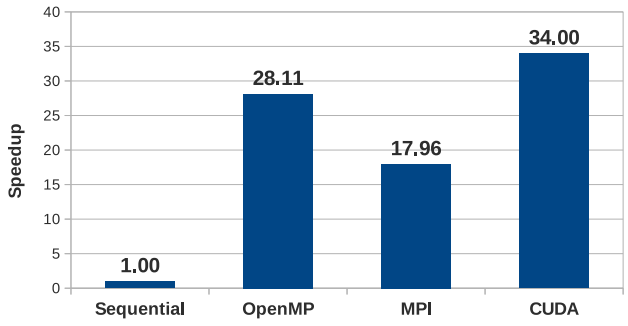


Fig. 12: Speedup results for the best version of each implementation, compared with the original implementation.

The best results were clearly obtained with the massively parallel implementation using CUDA. Yet, the shared memory version achieved a very similar value, specially taking into account that it could benefit from the latest most significant optimization performed in the CUDA implementation. Yet, this happens probably to the lack of a test case big enough to take advantage of all the hardware resources in the GPU, which was not possible to generate.

The distributed memory implementation achieved a speedup which is nearly half the other two versions. While this implementation was not subject to as many optimization work as the other two versions, it faces the same locality problems (although these are softened because the mesh is now divided among the computational nodes) and adds the communication overhead. This overhead easily becomes the bottleneck of this implementation, specially when using many computational nodes with only a few processors.

8 Conclusion

In this report, the analysis of the `polu` application and the successive attempts to optimize and parallelize it were presented.

The original implementation presented several performance problems, the most troubling being the structures implemented as `Arrays-Of-Pointers`. It also presented features which were not fully implemented or not targeted for improving the results of the computation. These features were removed. The main optimizations performed in the sequential implementation involved changing how the structures were implemented to `Arrays-Of-Structs`, and then to `Structs-Of-Arrays`, which achieved the best results (almost 10 times faster). Dependencies existed in the original code, which were removed along with the discarded features and some code adaptations.

Several approaches to parallelism were described. The first implementation, shared memory, used the OpenMP interface to parallelize the two core functions. It achieved a nearly perfect load balance but remained, just like the sequential version, highly limited by the lack of locality in the mesh structure. The `Structs-Of-Arrays` version also achieved the best results with this implementation (28 times faster than the original version). Intel® Hyper-threading technology was proved to be harmfull for the application performance.

The second implementation described in this document uses distributed memory with MPI. The main problem encountered while developing this implementation was found in the mesh partitioning scheme. While other options promised better results, time restrictions could prevent from achieving a functional version. A naive approach was taken, which optimized the computational workload while neglecting the communication overhead. The communication became the overhead of the program, surpassing locality. The best results were obtained using the two complete nodes, taking advantage of hyper-threading. Such can be explained by the existence of separate memory banks and the smaller partitions.

Lastly, CUDA was used to implement a massively parallel approach using GPUs. This implementation and the shared memory one were very similar, and took advantage of the same simplifications and optimizations. Two versions were designed: a naive one, obtained by translating the sequential code to CUDA and implementing some optimized kernels (a reduction which was later removed by the simplifications); and a second optimized version, which aimed to reduce the number of memory accesses and branch divergences. This last version obtained the best results in the entire project (34 times faster than the original implementation).

While CUDA obtained the best results, a more significant difference was expected. Further testing with larger test cases

is required to fully take advantage of the hardware capabilities. Yet, such case could not have been generated in useful time.

Among the several problems encountered while developing this code, the lack of locality of the mesh was the most relevant. It affected all the implementations here described. Works of other authors were found, and libraries exist to improve this issue, but their application was discarded due to time constrictions. These approaches are left as future work.

REFERENCES

[1] C. Geuzaine and J.-F. Remacle, "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities," Jun. 2012. [Online]. Available: http://www.geuz.org/gmsh/

[2] C. E. Leiserson, "Course 6.884 – lab 3: Stencil computing," Massachusetts Institute of Technology, February 2010, concepts in Multicore Programming. [Online]. Available: http://courses.csail.mit.edu/6.884/spring10/labs/lab3.pdf

[3] S. W. Williams, A. Waterman, and D. A. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-134, October 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html

[4] S. Williams and D. Patterson, "The roofline model: a pedagogical tool for program analysis and optimization," EECS Department, University of California, Berkeley, 2008. [Online]. Available: http://www.cs.berkeley.edu/~samw/research/talks/parlab08_roofline.pdf

[5] H. Hoppe, "Optimization of mesh locality for transparent vertex caching," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 269–276.

[6] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[7] J. Gilbert, G. Miller, and S. Teng, "Geometric mesh partitioning: Implementation and experiments," in *Parallel Processing Symposium, 1995. Proceedings., 9th International*. IEEE, 1995, pp. 418–427.

[8] C. Walshaw and M. Cross, "Mesh partitioning: a multilevel balancing and refinement algorithm," *SIAM Journal on Scientific Computing*, vol. 22, no. 1, pp. 63–80, 2000.

[9] NVidia, "Nvidia developer zone - gpu computing documentation." [Online]. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation

[10] *Intel® Xeon® Processor 5600 Series Datasheet Volume 1*, Intel Corporation, June 2011, revision 002. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-5600-vol-1-datasheet.pdf

[11] *Tesla™ M2050/M2070 GPU Computing Module*, NVIDIA Corporation, April 2010. [Online]. Available: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf

[12] *Dual-Core Intel® Xeon® Processor 5100 Series Datasheet*, Intel Corporation, revision 003. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/datasheet/313355.pdf

[13] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, March 2012.

[14] ——, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2011.

APPENDIX A
ENVIRONMENTAL SETUP

The tests described in this document were performed using a very specific subset of nodes from the SeARCH[4], here referred as SeARCH Group Hex.

Nodes in SeARCH Group Hex have two hex-core processors (with Intel® Hyper-threading technology) and 12 to 48 GB of RAM.Further detail regarding the hardware of these nodes can be found in table 2.

These nodes were used for every test performed in this document. Sequential, shared memory and GPU tests relied used only one node, where tests for distributed memory used two distinct nodes. Inside this group, tests were only limited to the subset with the highest processor clock frequency, not making any distinction between the nodes. The only exceptions are the GPU tests which used a very specific node with a Tesla M2070 GPU computing module. Further detail regarding this GPU module can be found in table 3.

Processors per node:	2
Processor model:	Intel® Xeon® X5650
Cores per processor:	6
Threads per core:	2
Clock frequency:	2.66 GHz
L1 cache:	32 KB + 32 KB per core
L2 cache:	256 KB per core
L3 cache:	12 MB shared
RAM:	12 to 48 GB

TABLE 2: SeARCH Group Hex hardware description. See [10] for further detail about this processor.

GPUs per node:	1
Model:	Tesla™ M2070
CUDA cores:	448
Peak (double):	515 Gflops
Dedicated Memory:	6 GB
Bandwidth:	148 GB/s

TABLE 3: SeARCH Group Hex GPU computing module hardware description. See [11] for further detail about this GPU.

Results obtained in a second type of nodes are shown in this document only to present a roofline. While this roofline should refer to the SeARCH Group Hex, this being the environment used for comparison, in previous stage of this project the `PAPI` library could only be used in a very specific node of SeARCH Group 201. Build the roofline for the Group Hex nodes, which meant rerunning tests with the `PAPI` library, was not possible in this stage due to time constrictions. Attempts were made in previous project stages, but the STREAM benchmark[5] returned less memory bandwidth than in Group 201, which is not true. Later analysis showed the problem was in the fact that such benchmark must not be run in Group Hex nodes using all the supported parallelism (6 threads, instead of 24, achieved around 15 GB/s).

SeARCH Group 201 is composed by nodes with two dual-core processors and 4 GB of RAM. Further detail regarding these nodes can be found in table 4.

4. http://search.di.uminho.pt
5. http://www.cs.virginia.edu/stream/

Processors per node:	2
Processor model:	Intel®Xeon®E5130
Cores per processor:	2
Threads per core:	1
Clock frequency:	2.00 GHz
L1 cache:	32 KB + 32 KB per core
L2 cache:	4 MB shared
L3 cache:	N/A
RAM:	4 GB

TABLE 4: SeARCH Group 201 hardware description. See [12] for further detail about this processor.

APPENDIX B
EXPERIMENT METHODOLOGY

All the tests performed in this document were performed considering only the core functions of `polu`'s main loop. Preprocessing stage and any cleanup operations were neglected, as their impact on the execution time significantly decreases as the time of simulation is extended.

Each test represents a total of 10 executions limited to 5000 iterations, where the final value is accepted to be the median, thus avoiding the influence of uncommon peak results.

This document shows only the results for the largest test case, which has a size of 62 MB. While the generator used in this project (`gmsh`) is able to generate meshes with thiner granularity, thus increasing its computational load, the conversion tool provided with `polu` is too complex to convert the generated mesh in usable time.

During the development of this project several tests were performed, with a special attention to some detailed measurements using the `PAPI`[6] library to use the available hardware counters. Yet, in this final stage, such results are not shown and focus is given only to final speedups.

APPENDIX C
ROOFLINE

The roofline model used in this document was prepared according to the guidelines in [3], except for the memory bandwidth roof and some adaptations. Compared to operational intensity, computational intensity (or arithmetic intensity) is a more complete measure for the program studied in this document.

As stated in appendix A, the roofline model presented in this document refers to a very specific node of SeARCH Group 201.

According to [12], [13], [14], these nodes' micro architecture is Intel® Core™, which is capable of decoding up to five instructions per cycle, has a throughput of up to 4 instructions per cycle and three full arithmetic logical units, where each has a throughput of one instruction per cycle for many kinds of computational instructions. This binds the peak throughput of computational instructions at 3 per cycle.

Since each core has a peak throughput of 3 computational instructions per cycle, each processor has 2 cores at a clock frequency of 2.0 GHz, and each node has 2 processors, this results in a peak of

$$3 \times 2 \times 2 \times 10^9 \times 2 = 24 \text{ GInstructions/s} \ . \quad (1)$$

This value corresponds to the CPU roof.

Although [3] recommends using "a series of highly tuned versions of the STREAM benchmark", the creation of such versions is out of the scope of this project. As such, the peak memory bandwidth was measured running the original STREAM benchmark in a SeARCH Group 201 node. The benchmark returned a peak value of 4.78 GB/s, which is the memory roof.

As for CPU ceilings, these were calculated decreasing the number of cores used, first by using only one processor (half the peak), then by removing thread-level parallelism (one core, a quarter of the peak). As measured in the previous report, this algorithm already presents itself with a high balance of floating-point multiplications and additions (two thirds of the TLP ceiling, as one of the ALUs would remain idle). The last CPU ceiling remaining would mean removing all the instruction-level parallelism (half the Mul/Add balance ceiling, only one ALU active).

Lastly, the absence of dual channel was used as the only memory ceiling. The influence of any other mechanism such as prefetch or unit stride accesses would have to be properly measured with the recommended tuned versions, which are not available for this document.

Figure 13 shows the resulting roofline model for the nodes of SeARCH Group 201.

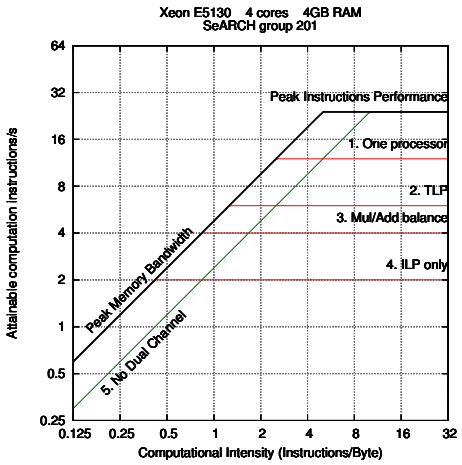


Fig. 13: Roofline for SeARCH Group 201.

6. http://icl.cs.utk.edu/papi/