Mesh Traversal and Sorting for Efficient Memory Usage in Scientific Codes

Pablo Barrio, Carlos Carreras
Dpto. Ingeniería Electrónica, E.T.S.I. Telecomunicación, Universidad Politécnica de Madrid
Ciudad Universitaria s/n, Madrid, Spain
{pbarrio, carreras}@die.upm.es

Abstract

Applications that operate on meshes are very popular in High Performance Computing (HPC) environments. In the past, many techniques have been developed in order to optimize the memory accesses for these datasets. Different loop transformations and domain decompositions are commonly used for structured meshes. However, unstructured grids are more challenging. The memory accesses, based on the mesh connectivity, do not map well to the usual linear memory model. This work presents a method to improve the memory performance which is suitable for HPC codes that operate on meshes. We develop a method to adjust the sequence in which the data are used inside the algorithm, by means of traversing and sorting the mesh. This sorted mesh can be transferred sequentially to the lower memory levels and allows for minimum data transfer requirements. The method also reduces the lower memory requirements dramatically: up to 63% of the L1 cache misses are removed in a traditional cache system. We have obtained speedups of up to 2.58 on memory operations as measured in a general-purpose CPU. An improvement is also observed with sequential access memories, where we have observed reductions of up to 99% in the required low-level memory size.

1. Introduction

Working with large datasets is a common requirement in HPC codes. Since the stress put at the memory increases with the size of the dataset, memory traffic is a major concern when porting codes to HPC systems. Modern CPUs include a complex hierarchy with many levels of caches, which has the effect of hiding the latency of the upper memory levels and preventing processor stalls to some extent.

However, although certainly making life easier to the programmer, cache mechanisms are not an efficient solution for every code, but a fairly good one that works for all. Caches do not account for access sequences specific to each code, but assume a benevolent time and space locality. In addition, novel devices used as accelerators, such as GPUs or FPGAs, are typically best suited to sequential memory access rather than cache-like random access. Unfortunately, the vast majority of the codes require random access.

This work targets the memory performance optimization in codes that operate on meshes. A mesh is a type of graph that organizes its vertices and edges around the concept of polygonal or polyhedral elements. Meshes are very popular in simulation environments as models to solve Partial Differential Equations (PDEs), such as some Computational Fluid Dynamics (CFD) or weather forecasting codes.

The data dependencies of a vertex mainly depend on that vertex and its neighbours. To a lesser extent, they may also access data from the neighbours of these first-level neighbours and consecutive neighbourhood levels. In a classical unoptimized access scheme, when a vertex V_o is processed, its data will be loaded to the computing element together with all of its neighbours V_1 to V_N. Every time that a vertex $V_i, i \in [1, N]$ is loaded, V_0 will also be requested, since it belongs to the neighbourhood of V_i. Hence, the scheme forces the transfer of each vertex as many times as its number of neighbours plus one.

We reduce memory traffic by rearranging the dataset so that the neighbours are kept close to each other in memory. The new vertex sequence affects the memory performance in the following scenarios:

- With random access as offered by traditional cache systems, the computing element takes advantage of implicit prefetching each time that a cache line is requested, minimizing or even supressing cache misses.

- When random access to the main memory is expensive or not available, as for some hardware accelerator boards, the dataset can be sequentially transferred between memory levels. The lowest-level memory must allow random access from the computing element, and it must be large enough to store the subset of the dataset being used at any given time during the mesh processing.

While the first of the above scenarios is easily understood, we will explain the second with an example. Figure 1 shows the high-level architecture of a system with one FPGA and one external memory, where random access to such memory is very expensive. Therefore, the vertices of the input mesh have been previously sorted. The serialized mesh is fed into the FPGA and each vertex is kept inside until all the computations involving its data are finished. Here, the fast on-chip memory is being used as a custom cache for the slower on-board memory. It is easy to see that the time lapse that a vertex can be kept inside the FPGA is limited by the size of the on-chip memory.

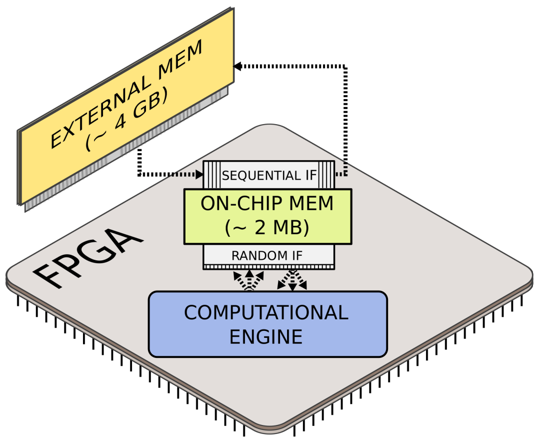


Figure 1. Example of a target system with one FPGA and one external memory

Although the access to the main memory is sequential, the algorithm may require random accesses, as Figure 1 shows. This is possible as long as the accesses are constrained to a vertex interval, which we refer to as *vertex window*. We define the *window* of a vertex as the portion of the mesh that is processed between the processing of the first neighbour of the vertex or itself —whatever comes first in the sequence— and its last neighbour. The maximum window size of a mesh must fit in the low-level memory. With a cache mechanism, failing to do so implies a time penalty for requesting the missing data to the higher level. In the example of Figure 1, there is no cache and thus missing data imply that the computational engine will stall indefinitely. Consequently, the size of the low-level memory imposes a limit to the size of the meshes that may be processed.

We propose a vertex sorting algorithm based on breadth-first traversing the mesh. This minimizes the maximum *vertex window* with respect to the original sequence, reducing the time lapse where the vertices must reside in low-level memory and consequently the required memory size.

The remainder of this paper is structured as follows. Section 2 reviews previous works related to mesh sorting and memory optimization. Section 3 presents a breadth-first method to sort the dataset for sequential memory access. Section 4 discusses the challenge of seed selection and its impact on the sorting quality. Finally, Section 5 summarizes the outcome of our research and further work to be done.

2. Related work

Mesh sorting has been widely proposed in other fields. In computer graphics, the data transfers between the CPU and the Graphics Processing Unit (GPU) are minimized. A common approach for triangular meshes is to divide it into *triangle strips*[1] that determine a particular vertex sequence, which is then sequentially fetched into the graphics pipeline. Strip processing greatly reduces bus traffic to the GPU compared to a random triangle-by-triangle approach, since the vertices are reused inside the GPU. OpenGL natively supports triangle strip formation since 1992.

Mitra and Chiueh [2][3] present an efficient mesh representation based on a breadth-first traversal. With the same motivation as the triangle strips, they exploit a different traversal method in order to maximize the usage of each vertex in several triangles. The breadth-first approach generates strips, although these wrap around the initial element. Hence, the vertices belong to more than two triangles inside the strip, allowing a wider use of the vertices in the future. This improvement comes at the cost of a larger buffer capable of storing more reusable vertices.

Graph traversal is intensively used in other computing disciplines such as artificial intelligence. Korf [4] analyzes the behaviour of several graph traversal algorithms with respect to memory. Zhou and Hansen [5] observe the growing size and shape of the frontier, concluding that BFS is more memory-aware than other popular graph search algorithms such as A* search.

Finally, modifying the sequence in which data are processed is suggested by Isenburg et al. [6] in their *processing sequences*. This work suggests traversing the dataset to limit the amount of stored data to the algorithm requirements. Implementing this idea requires that the target algorithm can be adapted to a fixed processing sequence; fortunately, mesh processing with nearest-neighbour dependencies falls into such category of algorithms.

3. Breadth-first processing order

Our breadth-first algorithm, BFsort for convenience, traverses the mesh starting from a seed vertex and building a spanning tree level by level. The neighbours of the seed form the first level, the second-generation neighbours will be the second level, and so forth. When a vertex is discovered, only two levels are being processed: the level that the vertex in process belongs to, and the next level with some of its neighbours. Any lower level is completely processed by the algorithm, and successive levels are not yet started. We take advantage of this property to reduce *vertex windows*.

```
1: for every $vertex$ do {Initialize}
2:     stat($vertex$) $\leftarrow UNDISCOVERED$
3: end for
4: $seq \leftarrow 0$
5: $seed \leftarrow$ select_seed()
6: $queue \leftarrow$ insert($seed$)
7:
   {Loop over the mesh}
8: while $queue \neq empty$ do
9:     $central \leftarrow$ extract($queue$)
10:    order($central$) $\leftarrow seq$
11:    $seq \leftarrow seq + 1$
12:
       {Insert neighbors in the queue}
13:    while at least one neighbor is not visited do
14:        $neighbor \leftarrow$ select_neighbor($central$)
15:        if stat($neighbor$) $= UNDISCOVERED$ then
16:            $queue \leftarrow$ insert($neighbor$)
17:            stat($neighbor$) $\leftarrow VISITED$
18:        end if
19:    end while
20:
21:    stat($central$) $\leftarrow FINISHED$
22: end while
```

Figure 2. Customizable BFsort

A comprehensive explanation of breadth-first algorithms can be found in [7]. In short, the algorithm starts from a seed, visiting all of its neighbours. Once the seed is finished, the algorithm takes the first processed neighbour and looks for neighbouring vertices that are not yet added. These are the second-level neighbours of the seed. The algorithm continues visiting and discovering vertices until the complete graph is traversed. As a guideline, Figure 2 shows the pseudo-code of our custom BFsort. Two functions serve us as customization points: `select_seed()` in line 5 and `select_neighbor()` in line 14. Seed selection is covered in Section 4. Neighbour selection, also called tie resolution, occurs when many vertices can be selected as the next in the sequence. We have found that different tie resolution strategies do not have an effect on vertex window sizes, but some of them improve differential compression schemes [8] suitable for particular target applications.

Figure 3 shows the evolution of the element-based traversal, as proposed by Mitra et al. [2][3], and the vertex-based breadth-first traversal as we propose. In short, the element-based scheme takes an initial triangle and adds its vertices to the sequence. Here we follow a clockwise vertex selection to choose the vertex priority inside a triangle. Subsequently, further neighbour triangles are chosen and continue filling the sequence with new vertices.

Similarly, the vertex-based scheme proceeds as follows. At each step, we define the *central* vertex as the one that triggers unvisited neighbours to be added to the sequence. We also define the *seed* as the first *central vertex* in the traversal, which is also the first vertex to be added to the sequence. We arbitrarily select vertex 0 as the seed. Starting from vertex 0, all of its neighbours are visited next and added to the sequence. We follow a clockwise tie resolution for clarity, but a random strategy would also be effective. Next, the first neighbour is selected –for example, vertex 6– and proceeds as the new central. This process iterates until every vertex is visited, resulting in the sequence 0-6-1-2-3-4-5-17-18-7-8-10-11-12-13-14-15-16-19.

The element-based approach is used by Mitra et al. to improve vertex reuse inside graphics pipelines. Nonetheless, we propose the second as being more efficient for scientific codes. The evolution of the sequence shows the step at which each vertex is eligible for deletion. This point is reached earlier for the vertex-based traversal; the element-based algorithm still requires processing five triangles (6-18-7, 7-8-1, 1-8-2, 2-3-0 and 0-3-4) to mark the first vertex, 0, as removable. Despite both being breadth-first based, they lead to different behaviours in terms of sorting.

When vertex 6 is selected as central, vertex 0 has finished its computations. Hence, it is not needed by the algorithm and can be discarded or written back to upper-level memories, freeing space in low-level memory. Generally, vertices are removed in the same sequence as they are discovered. As said, only the level in process and the next level must be visible from the processing element.

If the mesh is transferred sequentially, a random order will face the storage of the complete mesh in the worst case. This is, if the first and last vertices are neighbours, they will be brought together to the low-level memory, and the last will not be eligible for deletion until the end of the traversal. With BFsort, at any point, the open frontier is a small percentage of the complete mesh.

3.1. Window size results

Figure 4 presents the quality results of applying three sorting methods —original, element-based breadth-first and vertex-based breadth-first— to six test meshes. These are measured in terms of the maximum window length, which sets a lower bound on the required size of the memories.

The data for the original sorting yields two significant results. Firstly, sequential access to some meshes with the original order, as parsed from the mesh file, would require to store 99.9% of the mesh on-chip. Secondly, the window size and the mesh size do not behave similarly, which in turn makes difficult to set upper bounds for the sizes of the input meshes. Breadth-first achieves smaller frontier sizes than the original order, reducing the window by a factor of 5 and reaching in some cases a factor of 20.

Additionally, the vertex-based approach obtains smaller windows than the element-based one, as figure 3 predicted.

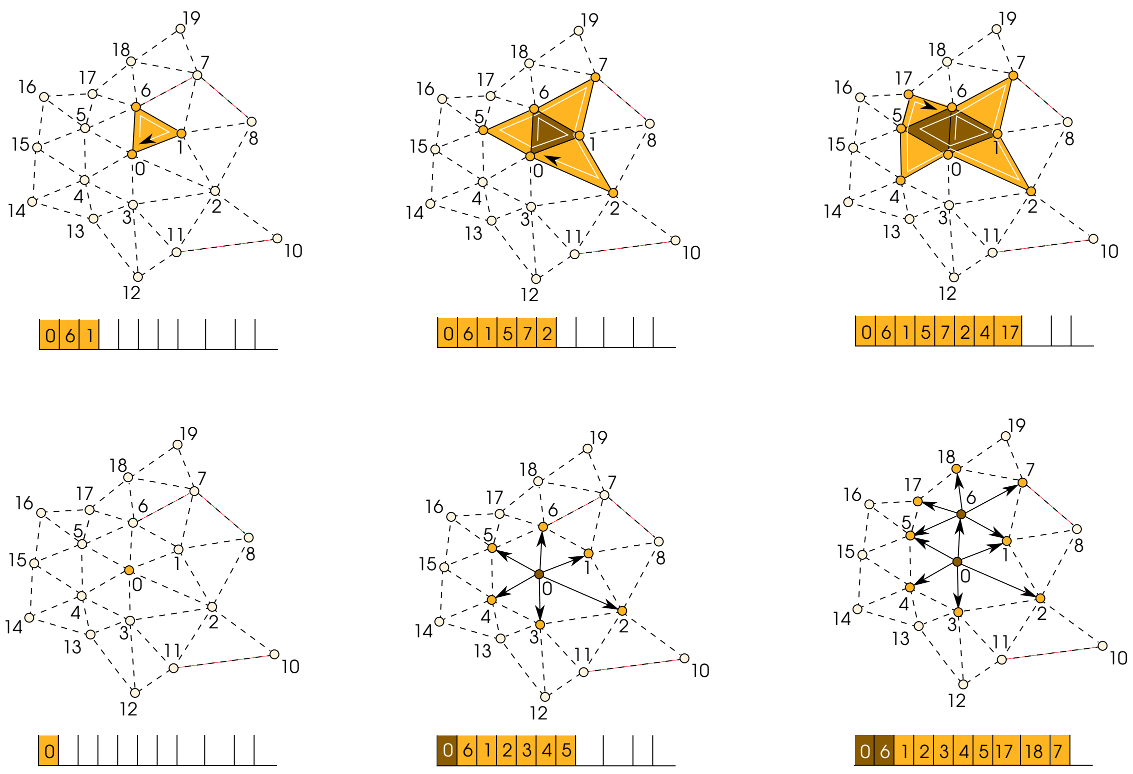


Figure 3. Breadth-first evolution by elements (upper line) and by vertices (lower line), together with the FIFO queues used to build the order. The lightly coloured vertices and elements must be stored in low-level memory, as well as the last dark-colored vertex. Previous vertices are eligible for deletion.

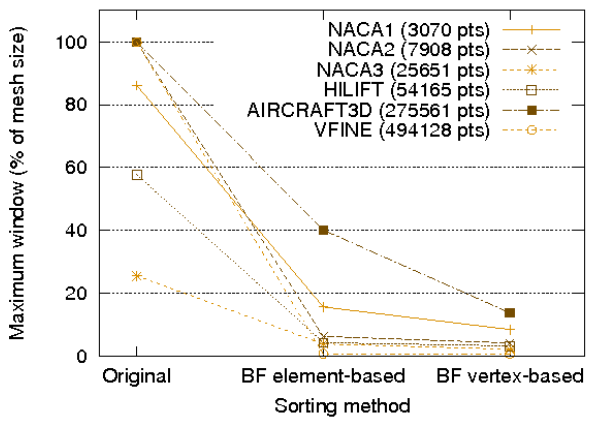


Figure 4. Maximum vertex window with different sorting methods, measured in number of vertices and percentage of the mesh size.

Both vertex- and element-based algorithms work well compared to the original sequence; however, the first one improves the results of the second by 1% to 25% of the mesh size. Hence, if the computational engine does not make use of elements such as triangles or tetrahedrals, a vertex-based sequence is more memory-efficient.

3.2. Performance on a traditional CPU

A detailed profiling has been carried out with traditional time analysis on a CPU and Cachegrind [9]. A simple test application converts the mesh into three arrays, each storing one coordinate of the vertices, and one adjacency array, storing the connections between vertices. The computation involves, for each vertex, adding up all of its coordinates and also the coordinates of its neighbours. Similarly to many common scientific codes, this computation generates nearest-neighbour dependencies.

Two instances of the program have been run. While the first instance computes the vertices in the original sequence, the second instance BF-traverses the data structures. Subsequently, the processing sequence is changed, and also the whole data structures are sorted in memory to guarantee that the cache block mechanisms behave well.

Table 1 shows the timings for both codes on an Intel Core2 Quad Q8400 at 2.66 GHz. The speedup on the mem-

		Execution time (s)		
Mesh	# vertices	Original	Vertex BF	S (%)
NACA1	3070	0.011	0.010	10.9
NACA2	7908	0.054	0.053	0.9
NACA3	25651	0.224	0.218	3.1
HILIFT	54165	0.501	0.467	7.4
PLANE3D	275561	5.936	3.716	59.76
VFINE	494128	5.682	5.092	11.6

Table 1. Real timings and speedups of the test code with two vertex sequences

		% D-L1 misses		% L2 misses	
Mesh	# accesses	Orig.	BF	Orig.	BF
NACA1	$1,1 \times 10^8$	5.2	4.9	0.0	0.0
NACA2	$2,8 \times 10^8$	5.3	4.0	0.0	0.0
NACA3	$7,4 \times 10^8$	7.9	7.7	5.3	4.9
HILIFT	$1,7 \times 10^9$	11.9	7.2	5.2	4.9
PLANE3D	$7,0 \times 10^{10}$	4.4	1.6	1.5	1.1
VFINE	$6,3 \times 10^9$	8.1	1.8	5.2	3.5

Table 2. Percentage of Data-L1 and L2 cache misses among all memory references. The sizes of the L1 and L2 caches are, respectively, 32 KB and 2 MB, 8-way associative with a block size of 64 B.

ory performance is set at around 10%, with two exceptions. While the NACA2 mesh obtains very poor speedups, PLANE3D, the only 3D mesh in the test set, runs 60% faster with the sorted mesh.

Unlike L1 misses (penalty $\sim$10 cycles), L2 misses are decisive in performance optimization (penalty $>$100 cycles). However, an interesting effect is observed in the tables. While PLANE3D shows the highest speedup, it is VFINE that achieves best results on cache miss removal. This is likely caused by the different ratio of computation to memory. Specifically, the computational load depends on the number of vertices, where VFINE beats PLANE3D, although the number of accesses to memory is one order of magnitude higher in the PLANE3D mesh. The reason is that the vertices of 3D meshes, such as PLANE3D, often have a higher number of neighbours, which implies a higher number of data dependencies. This fact also explains why VFINE, having such a high improvement on the cache performance, does not have an equivalent speedup.

4. Seed selection

The main disadvantage of sequentially accessing the mesh instead of using a cache mechanism is that the size of the memories imposes an upper bound to the maximum mesh size. Because the mesh is being transferred sequentially, the vertices belonging to a vertex window must be fully loaded onto the on-chip memory when that vertex is processed. If the memory size does not have enough capacity, the computational engine will stall indefinitely, as it has no means of requesting absent vertices to the upper memory level. Therefore, any technique that reduces the maximum vertex window allows for bigger meshes without requiring further costly techniques such as mesh partitioning.

Seed selection involves finding the vertex that will be used as the starting point for the BFsort algorithm. Seed selection is a separate problem itself; the BF algorithm will work regardless of the seed selection algorithm chosen. However, choosing specific seeds has proven to reduce the maximum vertex window. Previous work [10] has reported performance variations in a BFS-based algorithm due to different seed selections. However, to our knowledge, this is the first work that considers seed selection as a means to minimize the memory footprint of a breadth-first-sorted mesh.

In order to compare the possible variations between different seeds, we applied brute-force BFsort to all possible seeds, and measured the maximum vertex window obtained for each. Applying brute force to select the best seed gives best results but is not realistic. For small test meshes such as the NACA1 and HILIFT, the time to find the best seed is higher than the computing time of our test algorithm. As the number of vertices grows, the time to compute all the seeds grows exponentially. On the other hand, selecting the wrong seed may increase the memory usage by a factor of two. Hence, a low cost heuristic is needed.

For all test meshes, brute force has helped us identify where the good and bad seeds are located. These are unstructured meshes and, as such, they show regions with different concentrations of the vertices, making up a representative general test set. An appropriate seed selection is less important for structured meshes, since the maximum window sizes are similar for all seeds.

As an example, Figure 5 shows the meshed profile of an airplane wing, where the longest frontiers are marked for the cases where the best and worst seed have been used. The frontier obtained with the best seed gives a good idea of how the size of the maximum vertex window can be diminished:

1. Some areas of the mesh show higher concentration points. These areas, specifically the left and right sides of the wing, must be reached early in the traversal, since the frontier will be small at that point. As an example, the figure shows that the best seed is located near the concentration point to the right of the image.

2. The best seed is displaced to the left in order to start with one concentration point early in the execution. Besides, the maximum frontier occurs immediately after the right branch of the frontier has passed through that concentration point, but right before the

left branch passes through another concentration point. The conclusion is that dealing with various concentration areas simultaneously may result in an early growth of the frontier, which is not desirable.

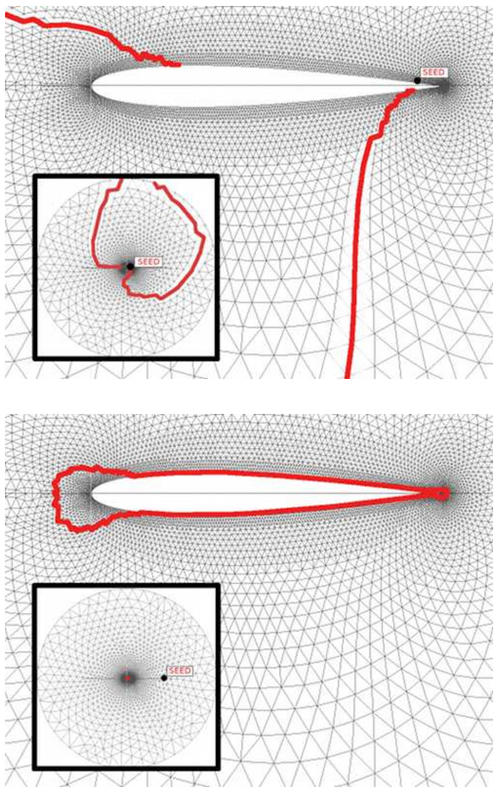


Figure 5. Maximum frontiers obtained with the best seed (top) and worst seed (bottom). The region of interest is shown in the main figures, whereas the boxed figures show the complete mesh.

On the opposite, the worst seed reaches the concentration areas at the end of the BF process. Furthermore, both areas are reached simultaneously, generating a frontier which is twice the size of the frontier for the best seed. The next paragraphs propose a few methods to select a seed for BF-sort. We measure the quality of these methods in terms of two metrics: the execution time and the quality of the seed compared to the best and worst seeds.

4.1. Center of mass (CM)

This approach is based on the vertex coordinates, taking its name from the calculation of the center of mass for a discrete solid body. The center of mass is calculated assuming that the weight of each vertex is 1. Equation 1 calculates the center of mass $\vec{CM}$ based on the position vector $\vec{r}$ of each vertex. Since the coordinates of the CM may not match up with any of the vertices, the closest vertex to the CM, using the Euclidean distance, is selected as seed.

$$\vec{CM} = \frac{\sum_{i=1}^{N} \vec{r}_i}{N} \tag{1}$$

Some improvements are observed when the weight of each vertex varies according to a given property. In particular, a vertex can be assigned a high weight if its neighbours are close to it, thus giving more overall weight to the concentration areas. The individual weight of a vertex receives the name of *vertex mass*. The calculation of the center of mass with variable vertex mass is given by Equation 2.

$$\vec{CM} = \frac{\sum_{i=1}^{N} m_i \vec{r}_i}{\sum_{i=1}^{N} m_i} \tag{2}$$

CM works well for simple meshes with no more than two concentration areas. The CM method will force the seed to be close to these areas, allowing them to be reached early in the BF-traversal process. Additionally, the CM will be displaced to the most concentrated area, which will be computed first. More complex meshes cause the CM method to fail, particularly when the vertices do not concentrate around points such as the wing front and rear in Figure 5, but around lines or curves. In addition, CM requires that each vertex has a position vector or coordinates, which is not mandatory for the general case of a mesh.

4.2. Bipartition and N-partition

Bipartition seed selection is based on the bipartition algorihm used to search in an ordered array. Figure 6 generates a temporary sequence and applies bipartition, searching through the vertex sequence for an appropriate seed.

A general bipartition search algorithm must operate on ordered sequences, since it is based on key comparisons of type “greater than” or “less than” to decide where to look next, to the right or to the left of the sequence. Likewise, the bipartition seed selection algorithm requires an initial temporary sort, as shown in line 2 of the algorithm. The key to be found is always the minimum value of the maximum window, and can only be known by computing its BF traversal, which is very costly. The bipartition algorithm reduces the number of required traversals from M to $log_2(M)$, with M being the number of vertices in the mesh.

Figure 7 shows different stages of the partitioning algorithm on two meshes. The starting search interval is initialized to the entire mesh, taking points 1, 2, and 3 as probes. The first iteration executes a complete breadth-first traversal at each probe, measuring the maximum vertex window. Because point 1 obtains better results than point 2, the interval between point 1 and 3 is selected. A new search is started

```
$seed \leftarrow$ random(size($mesh$))
$ordMesh \leftarrow$ BF_order(seed)
$startPt \leftarrow 0$
$endPt \leftarrow$ size($ordMesh$)

while $startPt < endPt$ do
    $midPt \leftarrow (startPt + endPt)/2$

    {Get max window for start and end points as seed}
    $ordMeshAux \leftarrow$ BF_order($startPt$)
    $startWindow \leftarrow$ get_window($ordMeshAux$)
    $ordMeshAux \leftarrow$ BF_order($midPt$)
    $endWindow \leftarrow$ get_window($ordMeshAux$)

    {Pick new interval to look for seed}
    if $startWindow < endWindow$ then
        $endPt \leftarrow midPt$
    else
        $startPt \leftarrow midPt$
    end if
end while
return $startPt$
```

Figure 6. Bipartition seed selection

with points 1, 3 and the middle point 4 as the probe set. The process continues until the search interval comprises only one point, which is selected as the best seed.

The lower figure shows a zoom into the last stages of a bipartition. As the search intervals get smaller, the vertex windows between neighbours become closer. At these stages, it is easier for the algorithm to make the right decisions, since the information is very local. The slope is easy to follow, and bipartition will make downhill movements until it finds a local minimum. On the contrary, during the first stages of the algorithm it is easy to make wrong choices, as the probe points are not close to each other thus not taking information of the vicinity into account.

Because the quality of the temporary sequence is very low, as shown in Figure 7, bipartition occasionally selects bad seeds. In order to minimize the chances to select a bad seed, an N-partition scheme may be used. Instead of partitioning the initial order in two, the N-partition algorithm divides it up into N chunks. As there are more probing points where the maximum window is calculated, the seed will be selected with better knowledge of the search space. However, N-partition requires to execute $log_N(M)$ BF traversals. Since $log_N(M) > log_{N'}(M)\ \forall N > N'$, we conclude that a high N in a N-partition scheme gives more accuracy than a low N at the cost of a higher computing time.

Finally, it is possible to save calculations at the start of the seed selection. In figure 7, we can identify a pattern that repeats periodically. If we choose the search interval as small as the period of that pattern, the search space is reduced compared with the complete mesh. In addition, the N-partition algorithm will be guided correctly to the best local seed. Furthermore, as the local minima do not vary greatly along the mesh (Figure 7), the selection of the particular interval among the complete mesh is unimportant as long as it covers the identified pattern period.

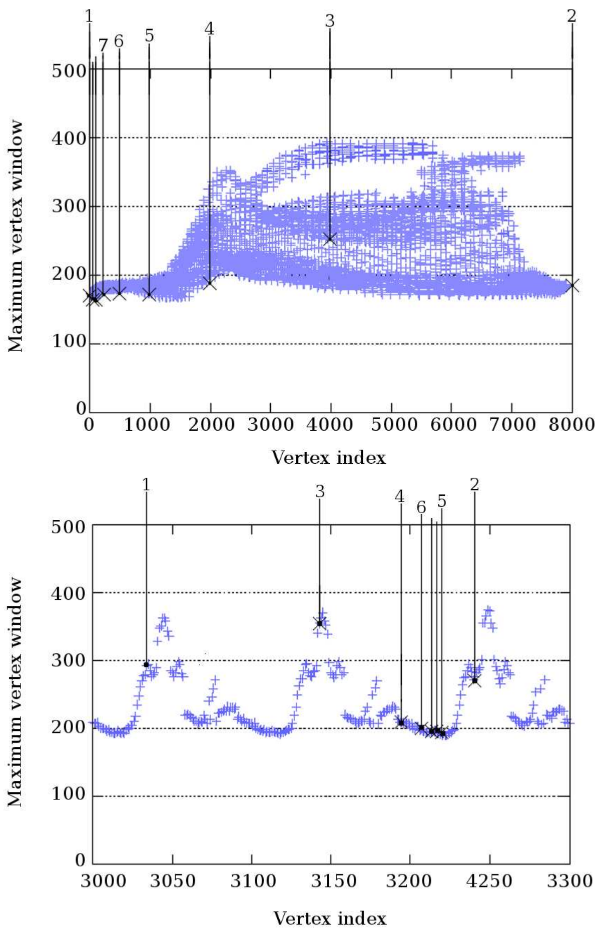
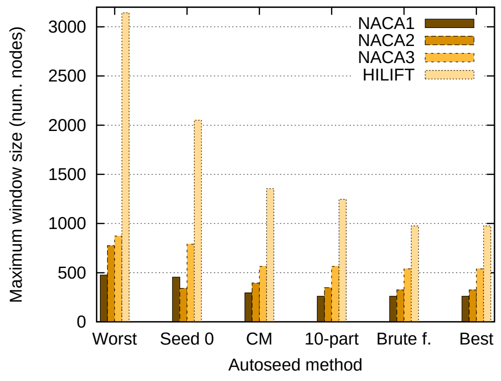


Figure 7. Execution of the bipartition algorithm. Overall view of the NACA3 mesh and a detailed view of mesh NACA2.

4.3. Seed selection performance results

Figure 8 compares several approaches to seed selection. Obviously, brute force always finds the best seed, but it takes several orders of magnitude longer than the *CM* method to execute. On the other hand, selecting a default seed such as vertex 0 gives potentially bad values and the maximum window size is not predictable.

The *CM* method, while keeping the execution time low, selects a good seed compared to the optimal selection, and constraints the maximum window size to a small percentage of the complete mesh. However, its correctness depends on the type of mesh: if several point concentration areas are present, or if they cluster around a line or more complex



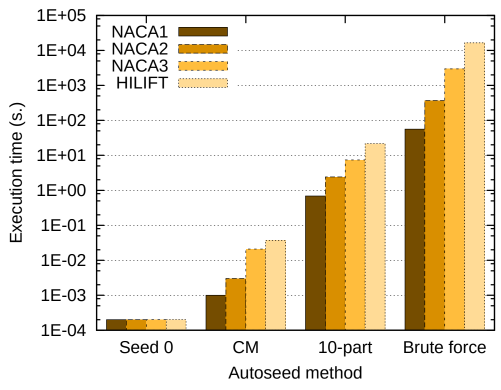


Figure 8. Comparison of the performance of seed selection methods

shapes, *CM* will not perform well.

N-partition finds better solutions than CM and, because it does not rely on the geometry of the mesh, it works for all meshes. However, greater computing times are required, since the calculation of maximum vertex windows requires to execute graph traversals for each probe point. Generally, we would consider N-partition only if the following processing stage is long. Also, if the mesh sorting application is independent of the HPC code, it will be possible to sort the mesh once and process it several times.

5. Conclusions

This work has presented a sorting method for meshes that is aimed at optimizing memory usage. The reduced memory footprint obtained allows us to remove cache misses. Our experiments show that this method can reduce the memory usage to a small percentage of the mesh, as opposed to the unpredictable behaviour of the traditional memory-oblivious computing sequences. We support that any algorithm that operates on meshes will benefit from a BF-sorted mesh, as long as the dependencies between vertices are similar to nearest-neighbour.

The effect of changing the processing sequence and the order of the mesh in memory has been presented together with the BFsort algorithm. Although cache-based systems may take advantage of the approach, we anticipate that an FPGA-based system will take even more advantage, as efficient sequential access to the high-level memory is possible.

Finally, we have identified seed selection as a problem that is worth solving in order to optimize memory performance. We approach the problem through CM, which offers a fast seed selection, and N-partition, which gives more accurate results, particularly with complex meshes.

Acknowledgments

This work has been funded in part by Airbus SAS under contract A8213093G and by the spanish Ministry of Science & Innovation under project TEC2009-14219-C03-02.

References

[1] F. Evans et al. Optimizing triangle strips for fast rendering. *Proc. Visualization ’96*, pages 319–326, 1996.

[2] T. Mitra and T. Chiueh. A breadth-first approach to efficient mesh traversal. *1998 SIGGRAPH Workshop*, pages 31–38, 1998.

[3] T. Mitra and T. Chiueh. An FPGA implementation of triangle mesh decompression. *Proc. FCCM ’02*, pages 22–31, 2002.

[4] R. Korf. Space-efficient search algorithms. *ACM CSUR*, 27(3):337–339, 1995.

[5] R. Zhou and E. Hansen. A breadth-first approach to memory-efficient graph search. *Proc. national conf. AI*, 21(2):1695–1699, 2006.

[6] M. Isenburg et al. Large mesh simplification using processing sequences. *Proc. IEEE Visualization ’03*, pages 465–472, 2003.

[7] J. Gross and J. Yellen. *Graph Theory and its Applications*. Chapman & Hall/CRC, 2006.

[8] C. Gotsman et al. Simplification and compression of 3D meshes. *Europ. Summ. School Princ. Multiresol. Geom. Model.*, 2001.

[9] Cachegrind: a cache and branch-prediction profiler. `http://valgrind.org/docs/manual/cg-manual.html`.

[10] M. Holzer et al. *Algorithms – ESA 2005*, chapter Engineering Planar Separator Algorithms, pages 628–639. Lect. Notes in Comp. Science. Springer, 2005.

[11] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, 2004.

[12] M. Herbordt et al. Computing models for FPGA-based accelerators. *Comput. Sci. Eng.*, 10(6):35–45, 2008.

[13] A. DeHon and J. Adams. Design patterns for reconfigurable computing. In *Proc. FCCM ’04*, pages 13–23, 2004.