Improvements to Hierarchical Pathfinding for Navigation Meshes.

Vahid Rahmani
Universitat Politecnica de Catalunya
Barcelona, Spain
rahmani@cs.upc.edu

Nuria Pelechano
Universitat Politecnica de Catalunya
Barcelona, Spain
npelechano@cs.upc.edu

ABSTRACT

The challenge of path-finding in video games is to compute optimal or near optimal paths as efficiently as possible. As both the size of the environments and the number of autonomous agents increase, this computation has to be done under hard constraints of memory and CPU resources. Hierarchical approaches, such as HNA* can compute paths more efficiently, although only for certain configurations of the hierarchy. For other configurations, performance can drop drastically when inserting the start and goal position into the hierarchy. In this paper we present improvements to HNA* to eliminate bottlenecks. We propose different methods that rely on further memory storage or parallelism on both CPU and GPU, and carry out a comparative evaluation. Results show an important speed-up for all tested configurations and scenarios.

CCS CONCEPTS

• **Computing methodologies → Planning with abstraction and generalization**; **Intelligent agents**;

KEYWORDS

Path finding, hierarchical representations, Parallel, CUDA

ACM Reference format:
Vahid Rahmani and Nuria Pelechano. 2017. Improvements to Hierarchical Pathfinding for Navigation Meshes.. In *Proceedings of ACM SIGGRAPH conference, Barcelona , Spain, November 2017 (Motion in Games 2017),* 6 pages.
https://doi.org/10.1145/3136457.3136465

1 INTRODUCTION

Path planning for Multi Agents in large virtual environments is a central problem in the fields of robotics, video games, and crowd simulations. In the case of video games, the need for highly efficient techniques and methods is crucial as modern games place high demands on CPU and memory usage. Typically it is not necessary to obtain the optimal path for all agents, but paths that look convincing.

The problem of path finding can be separated from local movement, so that path finding provides the sequence of cells to cross in the navigation mesh, and other methods can be used to set waypoints and to handle collision avoidance.

In this paper, we focus on abstraction hierarchies applied to path-finding to improve performance. A general notation consists of labelling the hierarchy as levels or layers in ascending order, with the lowest, L0 being the un-abstracted map in the game space and consecutive layers numbered L1, L2 and so on being the different levels of abstraction. The key idea consists on performing a search at a high level, which is then "filled in" with more refined sections of the path at lower levels, until a complete path is specified which can be followed by an agent [Bulitko et al. 2007].

Typically a high level solution can be rapidly calculated, and the challenge lies on inserting the specific Start (S) and Goal(G) positions to link them with the high level graph. Work in the literature shows that this inserting S/G step can become a bottleneck in both 2D grids [Botea et al. 2004] and Navigation Meshes [Pelechano and Fuentes 2016].

There are many techniques in the literature that have shown impressive improvements for the case of 2D regular meshes to increase speed without a large memory footprint [Sturtevant 2007]. However general navigation meshes consisting of convex polygons of different complexity, present more challenges given their irregular nature (i.e. not all the cells have the same size and edge length) [Van Toll et al. 2016]. In this work we propose several approaches to speed up the existing bottleneck in hierarchical path finding for general navigation meshes, and evaluate their advantages and limitations in terms of both memory usage and performance improvements.

2 RELATED WORK

Planning via hierarchical representation has been used to improve performance in problem solving for a long time [Sacerdoti 1974]. A two-level hierarchy can be created by abstracting the map into clusters such as rooms in a building or square blocks on a field [Rabin 2000]. An abstract action crosses a room from the center of an entrance to another, leading to fast computation at the cost of a non-optimal path. Hierarchical Path-Finding A* (HPA*) [Botea et al. 2004] reduces problem complexity on grid-based maps. The HPA* technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing each cluster are precomputed and cached. At the global (high) level of this method, an action consists of crossing a cluster in a single big step rather than moving to an adjacent atomic location and small clusters are grouped together to create larger clusters.

Visibility has also been used create hierarchical abstractions [Rabin 2000]. In this case, the graph nodes represent the corners of convex obstacles, and edges join nodes that can "see" each other (i.e. that can be connected with a straight line).

Hierarchical Navigation meshes have also been used to speed up path-finding [Pelechano and Fuentes 2016]. The method is based on a bottom-up approach to create a hierarchical representation using the multilevel k-way partitioning algorithm (MLkP), annotated with sub-paths information. Their approach is flexible in terms of both the number of levels in the hierarchy and the number of merged polygon between levels of the hierarchy.

Hierarchical Annotated A* (HAA*) [Harabor and Botea 2008] extends HPA* taking clearance into account. Kring and et al [Kring et al. 2010], introduced the Dynamic Hierarchical path-finding A* (DHPA*) and Static Hierarchical path-finding A* (SHPA*) hierarchical path-finding algorithms, along with a metric for comparing the dynamic performance of path-finding algorithms in games. In DHPA* the run-time cost is reduced by spending more time and memory usage in the build algorithm and less time in the search algorithm. In SHPA* the performance is improved and the memory requirements of HPA* are reduced. In DHPA*, improves the search performance by eliminating the time consuming "SG effort" that is present in HPA*. Our work is inspired by their method, but extended to the more general problem of navigation meshes where certain assumption such as cell size cannot be made beforehand.

The HNA* algorithm [Pelechano and Fuentes 2016] is a bottom-up method to create a hierarchical representation based on a multi-level k-way partitioning algorithm (MLKP) of a navigation mesh. Similarly to HPA*, HNA* also pre-computes sub-paths and stores them to be accessed by the on-line search algorithm.

In this paper we present several methods to solve the bottleneck that appears in HNA* when connecting the start/goal positions with the hierarchical representation.

3 THE HNA* ALGORITHM

The focus of this paper consists of solving the bottleneck that appears in HNA* when inserting start (S) and goal (G) positions into the high level abstraction graph. Before explaining the details of our approach, we would like to remind the reader the origin of this problem. A hierarchical navigation mesh consists of several layers, where a node of a higher level contains a group of merged nodes from a lower level. Finding a path in this representations consists of four steps (as illustrated in Figure 1): (1) insert S and G, (2) find path at high level, (3) extract sub-paths (stored from an off-line phase), and (4) delete S and G from high level graph. The bottleneck appears in step 1, since it is necessary to compute A* from S to each inter-edge in the high level node (inter-edges connect the high level node to its neighboring nodes). This cost increases rapidly with the number of inter-edges. And the number of inter-edges increases as we add more levels to the hierarchy or merge a larger number of polygons between levels of the hierarchy (for more details we refer the reader to the original paper [Pelechano and Fuentes 2016]). This effect has a negative impact on the overall performance of HNA* as it puts an upper limit on the performance benefits of the algorithm.

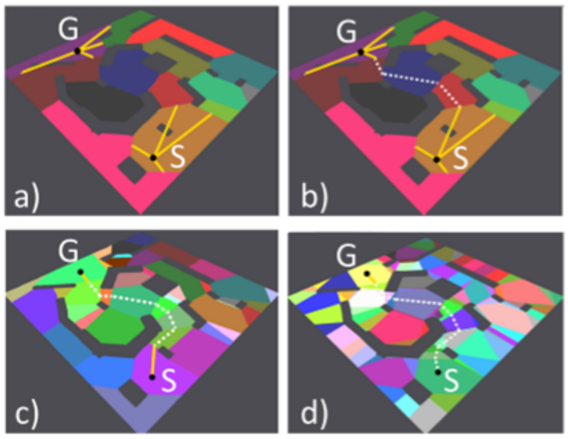

Figure 1: Path-finding computation: S and G are inserted and linked to their partitions at level 2 by calculating shortest paths to each portal in their respective node(a). Paths are calculated at level 2 (b), and then intra-edges are extracted from lower level 1 (c) and the final path is obtained for level 0 (d) [Pelechano and Fuentes 2016].

4 NEW INSERT S AND G APPROACHES

In this paper, we present three alternative solutions to solve this step and we carry out a quantitative evaluation of their advantages and limitations. The first solution focuses on storing further data, while the other two propose parallel implementations in both CPU and GPU.

4.1 Pre-Calculated Paths

The simplest way to solve this problem consist of pre-storing further information to speed-up the inserting step. We can calculate the A* path from a specific point p in each polygon at level 0 ($L0$ which corresponds to the original navigation mesh) to the inter-edges that appear in the higher level node of the hierarchy ($L\#$ where # represents the highest level). Therefore during the on-line phase it is only necessary to determine which polygon of L0 contains S, and extract the set of paths that connect p with the high level graph without the need to run A* between p and each inter-edge (from now on, since the algorithm is the same for both S and G, we will only refer to S).

Therefore the method includes an off-line and an on-line phase. In the off-line phase, the center point p_c of each polygon at L0 is calculated and the shortest paths and cost from p_c to the inter-edges in $L\#$ are calculated and stored in memory using a MultiMap hash table. Table 1 shows an example of such table, where we have 4 inter-edges for polygon 11, and thus for entry polyID=11 we can find 4 alternative paths with their corresponding cost. These paths would be the temporal connecting edges with the high level graph in order to compute A* at the higher level of the hierarchy during the on-line phase of the algorithm.

Therefore, when a new path search starts in the on-line phase, the algorithm checks the ID of the polygon containing S, takes its center position and extracts the temporal edges from the MultiMap table. We thus simplify the connect step with a fetch for the stored

Table 1: structure of MultiMap

Poly ID	Path	Cost
11	06-08-05-03	46.048
11	06-08-05-04	81.72
11	06-06	33.61
11	06-08	18.06
12	18-19-21-28	106.55
12	18-19-21-26	92.53

paths as opposed to computing A* on-line for each inter-edge of the node $n_{\#}^{S}$ (node of level # containing S).

Algorithm 1 shows the off-line phase of our method. Note that, for navigation meshes, it is necessary to compute the exact path from the center of each polygon, since we cannot assume that the shape and size of all cells is the same as it happens with 2D regular grids. It is important to note that center points are computed simply to obtain estimated distances to portals, when computing global paths. However this does not imply that the local movement of the agent has to cross the center point. Agents are steered towards the portal connecting with the next cell in their paths. Since all cells are convex, the path is free of collisions against the geometry).

Algorithm 1 Find-Path

```
procedure GET_PATH
    $N \leftarrow NumOfPolygons$                ▷ Number of polygons in L0
    $C \leftarrow NumOfCluster$                 ▷ Number of clusters in L1
    for i:=1 to N do
        $SId \leftarrow GetPolygonID[i]$
        $S \leftarrow GetPolygonCenterPos[i]$
        for k:=1 to C do
            $CId \leftarrow InterEdgeID[k]$
            if $SId == CId$ then
                $G \leftarrow InterEdgePos[k]$
                $(PolyId, Path, Cost) \leftarrow FindPathAstar(S, G)$
                $SavePath(PolyId, Path, Cost)$
            end If
        end for
    end for
```

4.2 Parallel Search on CPU

To exploit the parallel hardware architecture in depth, the algorithm should be adapted to run concurrently using multiple threads and shared memory access. The connecting S and G step is a highly parallelizable problem, as we can simply run each A* search in a different thread. In order to find a path from S/G in a polygon to their corresponding inter-edges using Multiple threads per polygon, we have used N threads concurrently to find an optimal path where $N = n + m$ with n being the number of inter-edges in $n_{\#}^{S}$ and m the number of inter-edges in $n_{\#}^{G}$. These threads work concurrently so that each thread calculates the optimal path from S or G to one of the inter-edges in the corresponding node $n_{\#}^{S}$ or $n_{\#}^{G}$. Our implementation uses the Boost library [BOO 2017].

4.3 Parallel search on GPU

The CPU usually contains several highly optimized cores for sequential instruction execution, while the GPU typically contains thousands of simpler but more efficient cores that are good at manipulating different data at the same time. In addition, the GPU has a memory system which is independent of that of its CPU. Such a design provides a higher bandwidth for accessing the global memory. In other words, cores of a GPU can retrieve and write data from/to the global memory much faster than a CPU [Zhou and Zeng 2015].

When several paths are being calculated in parallel in the Multi thread implementation, the Binary heap used for computing A* can become a bottleneck because it stores information in local memory. The A* search algorithm usually requires many accesses to global memory (especially in big scenarios) for storing and retrieving nodes from/to both open and closed lists. The A* algorithm also needs higher global memory bandwidth which can lead to a faster expansion rate during A* search.

In order to overcome this weakness and speed up the search process, we have used the GPU shared memory facility (using CUDA [NVIDIA 2017]). All the required data is stored into shared memory before any computation. Shared memory is much faster than local and global memory, because it is on-chip memory. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory.

A program designed to run on a GPU is called a kernel, and in CUDA the level of parallelism for a kernel is defined by the grid size and the block size [Nickolls et al. 2008]. One of the most important factors that can have an effect on parallelism performance is the degree of parallelism (DOP), which in our case corresponds to the number of inter-edges N (counting for both nodes of the high level graph containing S and G). We have defined a kernel with one block for the polygon containing S and another for G, plus n or m threads per block respectively.

5 EXPERIMENTAL RESULTS

5.1 Error and Memory Usage in Pre-calculated Path method

In this section we present the results achieved in terms of performance but also discus the limitations of each approach. For instance, the pre-calculated paths method, achieves the best performance, as we expected. However it requires additional memory and also introduces a small offset between the real position of S/G and the center position of each polygon.Therefore we need to measure the impact of both memory and offset in the results obtained. Figure 2 shows the memory usage in 5 different scenarios of a variety of sizes (shown as number of triangles in the original mesh).

Memory usage increases with the size of the scenario (Figure 2). The allocated memory for Dungeon scenario with 119 polygon is 2.9 MB while the allocated memory for Medieval City scenario with 16,867 polygons is 49.6 MB. Memory could be further reduced by storing in the hash table, only the next cell as opposed to the whole path. However this would require further accesses to the hash table, thus reducing performance.

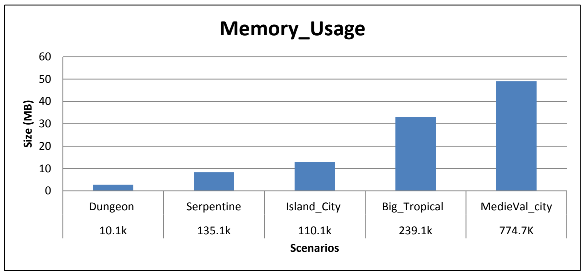

Figure 2: Memory usage in 5 different size scenarios.

In the pre-calculated paths approach, we have computed paths and costs from the center of each polygon to the inter-edges of its cell and store them in a hash table. When inserting new S and G points in any location of a polygon, the algorithm checks the hash table and fetches paths with the IDs that correspond to the polygons containing S and G positions. Undoubtedly, this introduces an offset between the center positions and the real S and G. However this offset represents only a marginal error when compared to the total length of the path (it simply adds a small offset at the beginning and at the end of the total path). However note that this offset simply affects the global path computation, and not the local path, as agents are not forced to walk through the center points. Our experimental results show a small impact on the total length of the path (3% on average for paths under 100m, and 5% on average for shorter paths).

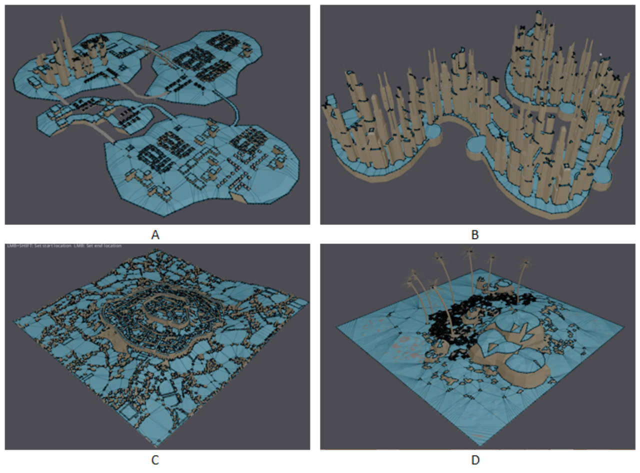

Figure 3: Different scenarios with their corresponding number of triangles in the mesh. A: City Island (110.3K), B: Serpentine City (135.1K), C:Medieval City (774.7K) and D: Big Tropical scenario (239.1K).

5.2 Performance Results for Pre-calculated paths method

For the evaluation of this method we have used several multilayer 3D scenarios as shown in Figure 3, with increasing numbers of cells in the original NavMesh and different hierarchical configurations. To compare the overall computational time of our pre-calculated paths method against HNA*, we have computed the average cost of calculating 100 paths with an Intel core i7-4770 CPU@3.5Gz, 16GB RAM. Results show that we can achieve significant speed-ups for all configurations, which was our ultimate goal.

For the City Island scenario, we can see in Figure 4-a1 the average cost of performing A* in this scenario is 2.2 ms. Figure 4-a1, shows that the performance of the Pre-calculated path method at $L1$ is not significantly faster than HNA*, this is due to the fact that at $L1$ the connecting S and G step does not represent an important bottleneck as can be appreciated in Figure 4-a2 . The strength of the new method can be observed for higher levels of the hierarchy. Figure 4-b1 and Figure 4-c1 show significant performance improvements when compared against HNA*. This improvements can be seen in Figure 4-b2 and figure 4-c2 where we have clearly manage to drastically drop the cost of the connecting S and G step.

Results are similar for the Big Tropical Island. The average cost of performing A* in this scenario is 1.7 ms. At L1, there is not a large performance gain, since the bottleneck of inserting S/G in HNA* is negligible. Our results show performance gain for all the values of μ tested ($\mu \in [2,20]$) at L1 with the fastest search being 1.12ms for $\mu = 20$. The advantages of the new implementation are noticeable for L2 and L3 after a specific value of μ. HNA* had a performance of 2.13ms for L2 and $\mu = 20$ and 9.01ms for L3 and $\mu = 10$ while our new HNA* obtained paths in 0.39ms for L2 and $\mu = 20$, and 0.25ms for L3 and $\mu = 10$.

Similar results where obtained for the Medieval city scenario (A* performance of 3ms). HNA* suffered from the insert S/G bottleneck after a specific value of μ. With 2.13ms in L2 and 9.01ms in L3 for $\mu = 10$ while our new HNA* had a computational time of 0.39ms in L2 for $\mu = 20$ and 0.25ms in L3 for $\mu = 10$.

5.3 Achieved Results of parallel search on the CPU

In order to evaluate our parallel CPU method, we have carried out experiments with the same set of scenarios and configurations. In parallel programming the performance of the method depends on the degree of parallelism of the problem to be solved (DOP), which in our case corresponds to DOP=N, with N being the number of inter-edges. The number of inter-edges can rapidly increase with the number of levels in the hierarchy and the number of merged nodes as shown in Figure 5 for the example of $L2$.

As we can see in Figure 6 with increasing DOP (number of inter-edges in our work) the total cost of our parallel CPU implementation decreases the connecting S and G step although eventually converging to a value. This is due to the fact that even though the increment of μ also increases the value of the DOP, the overhead of multi-threading outweighs the gains achieved.

From the results shown in Figure 6, we observe that the pre-calculated path and the Multi-threads implementation are much faster than the HNA* implementation on the CPU. However the pre-calculated path method still shows the most efficient results. HNA* and parallel CPU method exhibit similar results for small values of μ (i.e.while the number of inter-edges does not represent a big bottleneck in HNA*). However for larger values of μ, the cost

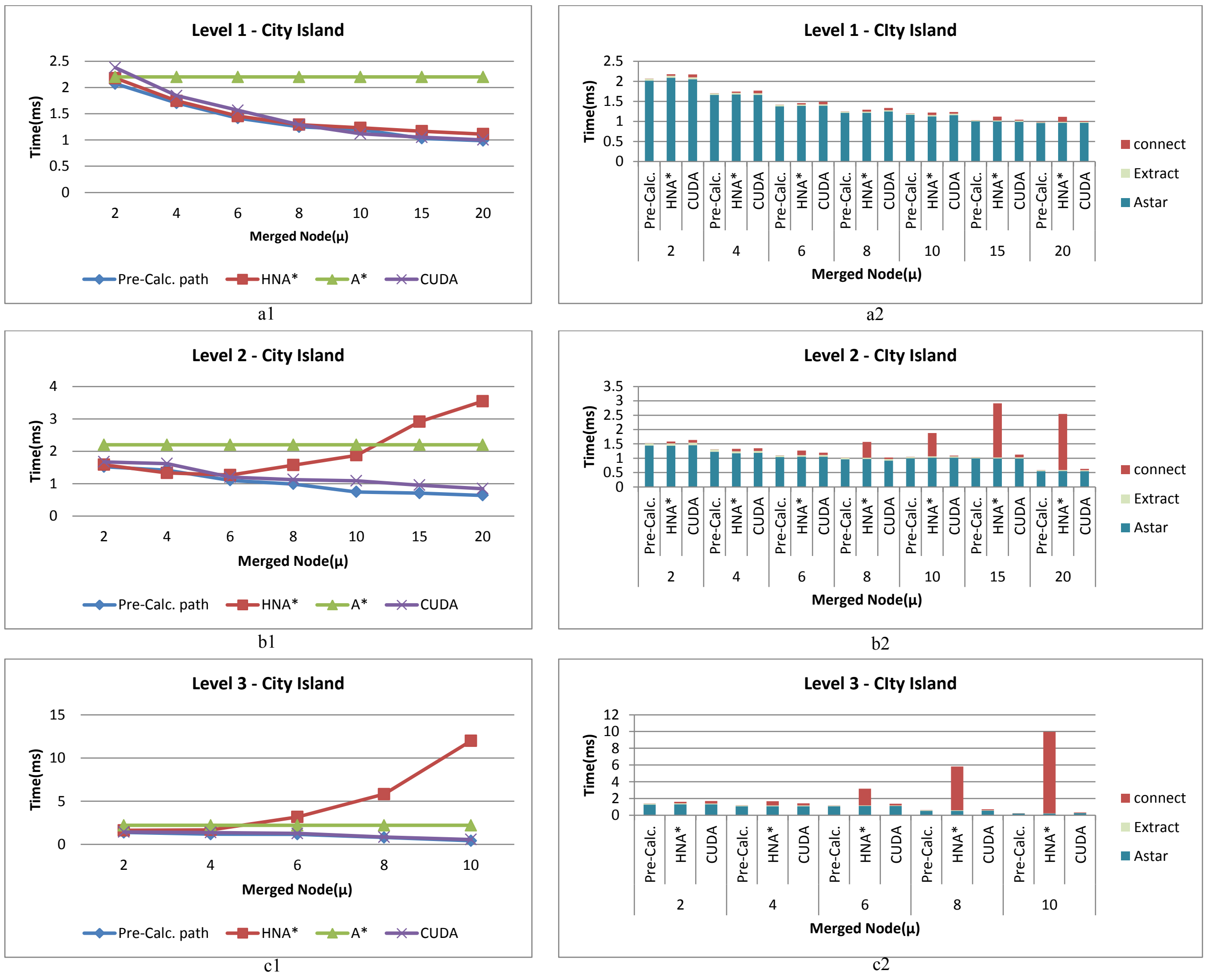

Figure 4: Performance results for the city Island.

of inserting S and G in HNA* can become increasingly expensive compared to pre-calculated path method or CPU parallel method. CPU parallel is more costly than pre-calculated because the binary heap used to implement the priority queue of A* can turn into a bottleneck in Multi thread implementations. The reason is that even though N (number of inter-edges per polygon) threads run in parallel, when it comes to inserting values in the binary heap, only one thread can remain active and all the other threads have to wait.

Finally, we also includ in this Figure 6 the results of connecting S and G with the parallel GPU version to compare it against the CPU version. We can clearly observe that the results for GPU are almost as efficient as the pre-calculated method.

5.4 Achieved Results of Parallel Search on the GPU

To compare the CUDA method against Pre-Calculated and HNA*, we have computed the cost of calculating 100 paths in the same

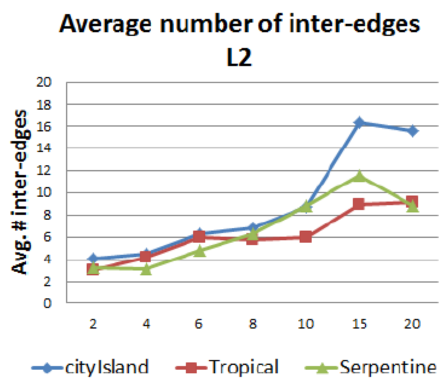

Figure 5: Average number of inter-edges for $L2$ as the value of μ increases.

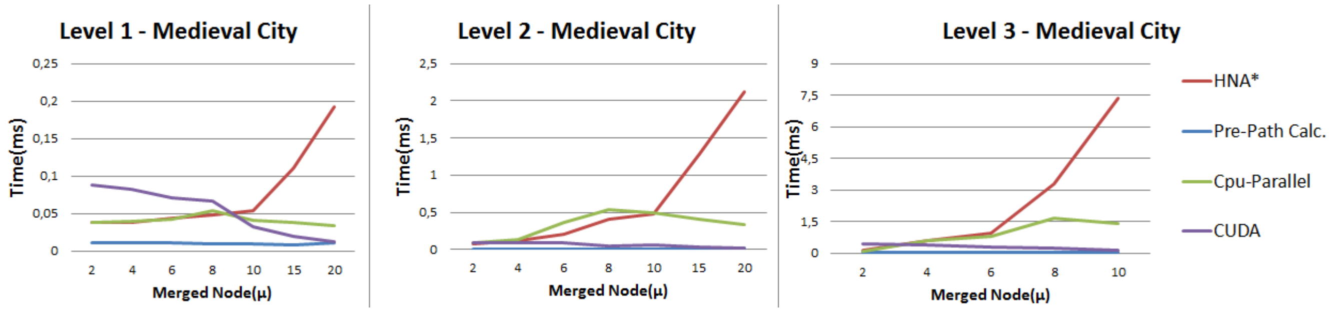

Figure 6: Performance cost for inserting S and G step with the parallel implementation on the CPU. Results shown the Medieval city scenario using a hierarchy of 3 levels.

scenarios and configuration (see Figure 4). The CPU used in these experiments is an Intel core i7-4770 CPU@3.5Gz with 16GB global memory. The GPU was a single NVIDIA Geforce GTX 420 with 2.4GB off-chip global memory and 2496 CUDA cores.

For the City Island scenario, Figure 4-a1 shows that the average cost of performing A* in this scenario is 2.2 ms. Figure 4-a1, for $L1$ of hierarchy and $\mu = [2, 20]$ the performance of Pre-calculated path method is faster than both CUDA parallel method and HNA*, which CUDA outperforming HNA*. As in previous experiments, the performance difference is not significant for L1, but for L2 and L3 it becomes highly significant. The time of computing a path for $L2$ and $\mu = 20$ is down to 0.648ms, and for $L3$ and $\mu = 10$ is down to 0.561ms for CUDA and 0.411ms for Pre-calculated path method.

As we can see in the right column of Figure 4, CUDA has a slightly higher cost when inserting S and G than the pre-calculated path. However the difference is negligible while saving memory footprint and avoiding the offset between S/G and the center the point of each polygon.

Similarly, for the Big Tropical scenario, the performance differences are not relevant for L1, but show drastic improvements from L2 onwards. For instance, the performance of CUDA in L2 and for μ=20 drops to 0.659ms while the performance of HNA* increases up to 2ms.058ms. However, Pre-Calculated paths is still faster than CUDA in L2 with the time being 0.396ms for μ=20.

Finally, we obtain similar results for the Medieval city scenario. In the original HNA* algorithm, the time of connecting S and G points increases up to 7.43ms in L3 for μ =10 whilst it drops to 0.14ms for CUDA, and 0.011ms for Pre-calculated path.

6 CONCLUSION

In this paper we have studied the problems of path-finding in large Scenarios for hierarchical representations based on navigation meshes. Our results have provided improvements over the basic HNA* algorithm. The first improvement that we have developed consists of the use of pre-calculated paths calculated from the center of each polygon in L0 (lowest level of the navigation mesh) to its inter-edges in the higher level of the hierarchy. Those paths are then stored in a MultiMap hash table and can be accessed efficiently during the on-line search. Given the highly parallel nature of our problem, the second improvement that we have implemented, consists of having a multiple threads version of HNA* algorithm on the CPU. In this implementation we have used threads in order to calculate paths concurrently each A* path between S/G and inter-edges of the high level node. Finally our third approach consists of a parallel version of HNA* on the GPU using CUDA. To evaluate our different methods we have used several multilayer 3D scenarios with increasing numbers of cells in the their navigation mesh and increasing number of merged polygons. Our results show that both the Pre-calculated Paths method and the CUDA version are faster than the original HNA* but Pre-calculated path method requires more memory usage than others. For all tested scenarios, the performance improvements are not very significant for L1, but they become very relevant from L2 onwards, as they eliminate the bottleneck of HNA* which was the connect S and G step. With this improvements, we have eliminated the important bottleneck from HNA* and thus obtain hierarchical path finding algorithm for general navigation meshes that offers speed-ups for a larger number of scenarios.

ACKNOWLEDGMENTS

This work has been funded by the Spanish Ministry of Science and Innovation and FEDER under grant TIN2014-52211-C2-1-R.

REFERENCES

2017. BOOST. (2017). "http://www.boost.org/".
Adi Botea, Martin Müller, and Jonathan Schaeffer. 2004. Near optimal hierarchical path-finding. (2004), 7–28.
Vadim Bulitko, Nathan R Sturtevant, Jieshan Lu, and Timothy Yau. 2007. Graph Abstraction in Real-time Heuristic Search. *J. Artif. Intell. Res.(JAIR)* 30 (2007), 51–100.
Daniel Harabor and Adi Botea. 2008. Hierarchical path planning for multi-size agents in heterogeneous environments. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium on.* IEEE, 258–265.
Alex Kring, Alex J Champandard, and Nick Samarin. 2010. DHPA* and SHPA*: Efficient Hierarchical Game Worlds. (2010).
John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (2008), 40–53.
NVIDIA. 2017. CUDA. (2017). http://www.nvidia.com/object/cuda_home_new.html.
Nuria Pelechano and Carlos Fuentes. 2016. Hierarchical path-finding for Navigation Meshes (HNA*). *Computers & Graphics* 59 (2016), 68–78.
S. Rabin. 2000. A* Speed Optimizations. *Game Programming* (2000), 272âĂŞ278.
Earl D. Sacerdoti. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5, 2 (1974), 115 – 135.
Nathan R Sturtevant. 2007. Memory-Efficient Abstractions for Pathfinding. *AIIDE* 684 (2007), 31–36.
Wouter Van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. 2016. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*. ACM, 91–100.
Yichao Zhou and Jianyang Zeng. 2015. Massively Parallel A* Search on a GPU. Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence.