Multi-agent parallel hierarchical path finding in navigation meshes (MA-HNA*)

Vahid Rahmani[a], Nuria Pelechano[a,*]

[a] *Universitat Politcnica de Catalunya, Barcelona, 08034, Spain*

ARTICLE INFO

Article history:
Received March 24, 2020

Keywords: Multi-agent path finding, hierarchical search, parallel path finding

ABSTRACT

One of the main challenges in video games is to compute paths as efficiently as possible for groups of agents. As both the size of the environments and the number of autonomous agents increase, it becomes harder to obtain results in real time under the constraints of memory and computing resources. Hierarchical approaches, such as HNA* (Hierarchical A* for Navigation Meshes) can compute paths more efficiently, although only for certain configurations of the hierarchy. For other configurations, the method suffers from a bottleneck in the step that connects the Start and Goal positions with the hierarchy. This bottleneck can drop performance drastically. In this paper we present two approaches to solve the HNA* bottleneck and thus obtain a performance boost for all hierarchical configurations. The first method relies on further memory storage, and the second one uses parallelism on the GPU. Our comparative evaluation shows that both approaches offer speed-ups as high as 9x faster than A*, and show no limitations based on hierarchical configuration. Finally we show how our CUDA based parallel implementation of HNA* for multi-agent path finding can now compute paths for over 500K agents simultaneously in real-time, with speed-ups above 15x faster than a parallel multi-agent implementation using A*.

1. Introduction

Path planning for multi-agents in large virtual environments is a central problem in the fields of robotics, video games, and crowd simulation. In the case of video games, the need for highly efficient techniques is crucial as modern games place high demands on CPU and memory usage.

Path finding should provide visually convincing paths for one or many autonomous agents in real time. Typically, it is not necessary to obtain the optimal path for all agents, instead use paths that look convincing to the viewer and can be computed within strict time constraints (to support 25 frames per second considering all other computations required in a game such as rendering, physics simulation, and AI).

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 way-points and to handle collision avoidance against other moving agents in the cell [1].

In this paper, we focus on abstraction hierarchies applied to multi-agent 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 representing higher levels of abstraction. The key idea consists of 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.

*Corresponding author: Tel.: +34-934-137-858;
e-mail: v.rahmani2015@gmail.com (Vahid Rahmani), npelechano@cs.upc.edu (Nuria Pelechano)

Typically a high-level solution can be rapidly calculated, and the challenge lies in inserting the specific Start (S) and Goal (G) positions to connect them with the high-level graph. The literature in this field shows that the S/G (Start/Goal) connection step can become a bottleneck in both 2D grids [2] and Navigation Meshes [3].

There are many techniques that have shown performance improvements for the case of 2D regular meshes without a large memory footprint [4, 5]. However, general navigation meshes consisting of convex polygons of different complexity present more challenges due to their irregular nature (i.e. not all the cells have the same size and edge length) [6]. In this work we propose two approaches to eliminate 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. The proposed solutions provide a large speed up for all configurations of the hierarchy, and makes our new HNA* algorithms viable for even larger environments than before. Our solution can also be combined with multi-agent simulation, to handle several hundred thousand agents computing paths simultaneously in real time.

2. Problem formulation

A world map is typically given as a polygon soup. In order to have agents navigating a world map, it is necessary to find a representation of the walkable space. This can be done with a navigation mesh, which represents the walkable space as a collection of convex polygons called cells (could be triangles or polygons of more than three sides), where borders between adjacent cells are called portals [7]. Agents can move within any two points of a cell or cross portals to move between adjacent cells, without colliding with the static obstacle borders of a cell. This representation can be expressed as a graph $G = (N, E)$, where the collection of cells or convex polygons are the nodes or vertices of the graph $N =< p_0, p_1, ..., p_n >$, and the portals are the edges E, with each edge e_{ij}, corresponding to the edge between two adjacent polygons p_i and p_j. The cost of an edge $c(e_{ij})$is calculated as the distance between the center of polygon p_i to the center of polygon p_j, and thus it is always a positive value. Path-finding involves finding a path $P = \langle S, ..., u, ..., v, ..., G\rangle$ which is a sequence of nodes connected by edges, from the starting position S to the goal position G. The cost of a path $c(P)$ is the sum of all the costs assigned to the edges along the path P, and since all edges costs are positive values, the cost of a path will always be a positive value. The shortest path between S and G is the path of minimum cost among all possible paths. A* performs an informed graph search, by computing for each node being explored the function $f(x) = c(x) + h(x)$, where $c(x)$ is the current cost from S to node x, and $h(x)$ is the heuristic that estimates the optimal cost of the path from x to G [8]. When dealing with maps, $h(x)$, can be computed as the Euclidean distance between the position of the center of node x, and the position of the center of node G. With this heuristic, A* can always find the optimal path, which is the path of minimum distance.

Each level of the hierarchy $Lx, x > 0$, is represented by a new graph G_x which is created by merging μ connected nodes from G_{x-1} (the value of μ is decided by the user). The new graph $G_x = (N_x, E_x)$, consists of a set of nodes $N_x = \langle n_x^0, n_x^1, ..., n_x^m\rangle$, where each node in G_x is a subgraph of μ connected nodes from G_{x-1}, so that $n_x^i = \langle n_{x-1}^j, n_{x-1}^k, ..., n_{x-1}^l\rangle$. Edges E_x in G_x are the subset of edges from G_{x-1} that connect two nodes n_x^s and n_x^d, where $s \neq d$.

Definition 2.1. An *Inter-edge*, ι_x^{sd}, in G_x is an edge e_{ij} from G_{x-1} that connects two nodes n_{x-1}^i and n_{x-1}^j, such that n_{x-1}^i is inside n_x^s, n_{x-1}^j is inside n_x^d, and $s \neq d$.

For those edges e_{ij} from G_{x-1} that connect two nodes n_{x-1}^i and n_{x-1}^j, such that both n_{x-1}^i and n_{x-1}^j are inside n_x^s, they become internal edges of node n_x^s. Therefore, there is no loss of connectivity between G_{x-1} and G_x, since all the set of edges in E_{x-1} are now either internal edges of nodes n_x^s in G_x or *inter-edges* in G_x.

These concepts are shown in Figure 1. In the case of L1, the merged nodes from L0 are polygons of the navigation mesh. Figure 2 shows an example of a simple navigation mesh from level L0 to L3. Colors are used to represent nodes at each level, so we can appreciate how each navigation mesh polygon turns into a node at L0, and then several connected polygons from L0 are merged in one larger node at L1, and similarly for L2.

The graph G_x contains a partition of G_{x-1}, with nodes at Lx being groups of adjacent nodes from $L(x-1)$, and edges E_x being a subset of the edges of E_{x-1}. Each node n_x can be traversed by finding an internal path between a pair of *inter-edges*. Such internal paths are represented by a sequence of polygons and can be pre-computed and stored.

Definition 2.2. An *Intra-edge*, $\pi_x^{s(dk)} = \langle p_0, p_1, ..., p_k\rangle$, is a sequence of polygons from G_0 that represent the optimal path to traverse a node n_x^s between two *inter-edges* ι_x^{sd} and ι_x^{sk}. Therefore, $\pi_x^{s(dk)} = optimalPath(\iota_x^{sd}, \iota_x^{sk})$. Its weight is computed as the sum of costs of the edges e_{ij} along the path, $c(\pi_x^{s(dk)}) = c(e_{01}) + c(e_{12}) + ... + c(e_{(k-1)k})$, where e_{ij} is the edge between nodes p_i and p_j.

A node n_x^s will have an *intra-edge* for each pair of *inter-edges*. In order to find a high level path, we need a Hierarchical Navigation Graph, $HNG_x = (V'_x, E'_x)$, which captures the connectivity of G_x given by the relationships between *inter-edges* and *intra-edges*. In HNG_x, the vertices are all the *inter-edges* in the partition represented by G_x, $V'_x = \langle \iota_x^{sd}, \iota_x^{dk}, ..., \iota_x^{lm}\rangle$, and the edges, E'_x are *intra-edges*, $\pi_x^{d(sk)}$ connecting each pair of *inter-edges*, for which a path exists.

Note that HNG_x maintains the connectivity of the navigation mesh, but in a more compact representation, where only some edges are kept as nodes in HNG_x (those *inter-edges*, which depend on the hierarchical level L and the merging factor μ), and the shortest paths at $L0$ between those nodes are precomputed as *intra-edges*. Therefore HNG_x is built in a way that guarantees that the connectivity between polygons at L0 is kept regardless of the hierarchical configuration.

If a path, $P_0 = \langle p_S, p_1, p_2, ..., p_G\rangle$, exists at G_0 , then there will be a path at level Lx. Computing path finding in HNG_x gives as a result the path $P_x(S, G) =$

$\langle \pi^S_{temp}, \pi^{s(dk)}_x, \pi^{k(sq)}_x, ..., \pi^{r((m-1)m)}_x \pi^G_{temp} \rangle$. $P_x(S, G)$ is the high level path. The temporal paths, π^S_{temp} and π^G_{temp}, were created during the connect S and G steps, which computes a path at level L0 for the subgraph represented by the high level node S, and similarly for G. Therefore $\pi^S_{temp} = \langle p_s, p_0, p_1, ..., p_n \rangle$ where p_n is a polygon with one of the edges being the *inter-edge* that connects p_n with the first polygon in $\pi^{s(dk)}_x$. Extracting the sequence of polygons from each *intra-edge* $\pi^{i(jk)}_x$ we obtain the full sequence of polygons to traverse the navigation mesh between S and G (Proof in appendix A).

3. Related Work

The most common approaches to speed-up path-finding, consist of either building some abstraction or hierarchy where path finding can be performed with smaller graphs (independently of the path-finding algorithm used), or else modifying the A* algorithm to gain speed by losing path optimality. Parallelism can then be applied to compute multiple paths simultaneously and thus provide real-time multi-agent path finding.

3.1. Hierarchical representations

Planning via hierarchical representation has been used to improve performance in problem solving for many years [9].

Holte et. al. [10] introduced hierarchical A* to search in an abstract space and use the solution to guide the search in the original space. There has also been work on abstraction based on bottom-up approaches for general graphs [11]. Sturtevant and Jansen [12] provided examples of a number of different abstraction types over graphs. In this work graphs were created from 2D grid-like structures by setting a node for each walkable cell.

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 [13]. An abstract action crosses a room from the center of an entrance to another, leading to fast computation at the cost of obtaining a non-optimal path. However, this representation requires some semantics to build the abstraction level, and thus cannot be used for any input map. When building an abstraction or hierarchical representation to speed up path finding, there is typically a penalty in terms of path optimality. Bulitko et al [14] showed that the quality of paths can decrease exponentially with each level of abstraction. Sturtevant and Geisberger [15] studied the combination of abstraction and contraction hierarchies to speed up path-finding.

Hierarchical Path-Finding A* (HPA*) [2] 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 pre-computed 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. Hierarchical Annotated A* (HAA*) [16] extends HPA* taking clearance into account. Therefore, it allows for different agent sizes to efficiently plan high quality paths in heterogeneous-terrain environments. Sturtevant and Buro [17] presented Partial-Refinement A* (PRA*) which, similar to HPA*, builds a hierarchical representation of a 2D regular grid to perform a quick abstract search and then refines sections of the path. Their hierarchy groups four connected cells into *4-cliques*, and then adds *orphan* nodes when possible (single cells only accessible trough one edge), instead of simply overlapping a lower resolution grid. Pre-computing sub-goals for grids has also been widely used to lessen the effort of finding paths online. However such pre-computation can be time consuming and thus some authors have proposed solutions to avoid pre-calculating paths, and use dynamic programming instead[18]. Their method builds a partition based on reachability which avoids revisitation.

Kring et al [19], 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.

Navigation meshes offer a better representation of the walkable space and a better fit to the contour of obstacles [20, 6]. To build polygonal navigation meshes, the tool Recast [21] is currently included in popular game engines such as Unity3D and Unreal Engine, and thus used in many recent games. Some authors have used navigation meshes overlapping 2D regular grids to perform a multi-domain planning at different granularities[22]. This approach speeds-up path finding over the regular grid by applying the tunnel algorithm with the solution obtained at the navigation mesh level. However, their highest level of abstraction is the navigation mesh itself. In order to provide faster path finding also for navigation meshes, HNA* [3] presented an approach that could handle the complexity of such a representation. This 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 (known as *intra-edges*). HNA* is flexible in terms of both the number of levels in the hierarchy and the number of merged polygon between levels of the hierarchy. The advantage of MLkP is that it provides a balanced number of both walkable cells and connections between partitions (known as *inter-edges*). Figure 1 shows an example of a hierarchical Navigation Graph (HNG) for a two-levels-hierarchy and $\mu = 4$ (where μ is the number of merged polygons).

Both HPA* and HNA* suffer a bottleneck when the nodes in the hierarchy are very large, which increases the number of *inter-edges*. The larger number of *inter-edges* has a direct impact on the first step of both methods, which consists of computing A* from the Start and Goal positions to all the *inter-edges* of the high level node were they lie. DHPA* [19], improves the search performance by eliminating the time consuming "SG cost" that is present in HPA*. Our work is inspired by their method, but extends it to the more general problem of navigation meshes where certain assumption such as cell size cannot be made beforehand.

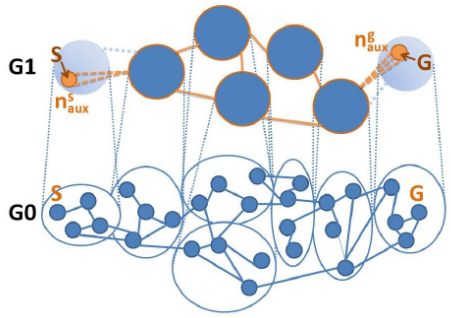


Fig. 1. Example of HNG with two levels and $\mu = 4$. The orange circles and discontinuous links represent the temporal nodes and edges created after linking Start and Goal points to the HNG. This temporal graph is where the HNA* runs [3].

*3.2. Variants of A**

Ammar et al [23] presented the RA* algorithm which is a new linear time relaxed version of A*. This method proposed to solve the path finding problem for large scale grid maps. The main goal of this algorithm is to find an optimal or near optimal path with small deviance from the optimal solutions by spending much smaller execution times than traditional A*. This method exploits the grid map structure to build a highly accurate approximation of the optimal path, without visiting any block more than once (unlike A* for which the cost $g(n)$ of a node n may be computed more than one time.)

Another variant of A* search algorithm is the Theta* algorithm [24]. The fundamental Theta* calculation is an existing algorithm that produces near-optimal results for a running time near A* around 8-directional grids. The main problem of this algorithm is that it often finds paths with unnecessary turns. In [25], Shunhao Oh et al. have shown that by restricting the search scope of Theta* algorithm to taut paths, the algorithm can provide much shorter paths than the basic algorithm.

Parallelism can provide a performance boost. Some researchers have proposed methods based on parallelising a single path search (typically losing path optimality just like with hierarchical approaches [26]), whereas other researchers have kept A* and applied parallelism to compute multiple agents' paths simultaneously. Most of the effort focuses on finding the best way to distribute work and syncing tasks to make the most of the GPU kernels and threads, while managing correctly memory accesses. Caggianese and Erra [26] proposed a parallel version of A* using a grid map decomposition and CUDA, and obtained results that run faster than a GPU implementation of Real-Time Adaptive A* (P-RTAA*). Merrill et. al presented a parallelisation of BFS (Breadth First Search) tailored to the GPUs requirement for large amounts of fine-grained, bulk-synchronous parallelism [27]. Ortega-Arranz et al [28] presented a parallel implementation of Dijkstras algorithm, which achieved between 13x and 220x speed up compared to the CPU sequential Dijkstras algorithm. Caggianese et al [29], proposed an A* implementation for the GPUs, based on planning block (P-BA*). This method is suited for grid based maps. First the search space is subdivided into small regular regions called tiles and then a parallel search is performed.

3.3. Multi-Agent Path Finding

There has been work focusing on Multi-Agent Path Finding (MAPF). Bounded Multi-Agent A* (BMAA*) [30] is a real time heuristic search algorithm for multi-agent path-finding. Li et al [31], proposed a LAMAPF method (MAPF for large agents path finding) which is an adapted version of Conflict-Based Search (CBS), to solve LA-MAPF, called Multi-Constraint CBS (MCCBS). The MCCBS adds multiple constraints instead of one constraint for an agent when it generates a high-level search node.

To run searches for thousands of agents simultaneously in [29], Caggianese et al. exploit the fact that, given a set of points as start positions and one goal position, it is likely that the explored paths share sub paths. The method tries to determine these shared sub paths by computing simultaneously all potential sub path types inside the planning blocks and considering that the sub path should converge toward the goal position. The method has two steps: border-to-border search and start-to-border search. This method is thus limited to all agents sharing the goal position.

Parallel computing for multi-agent path finding has also been used for other types of navigation meshes, such as 2D regular grids [32], and triangulations [33], where the approaches are strongly dependent on the specific implementation of the grid or triangulation. Therefore neither solution can be easily applied to a general navigation mesh given as an input.

The purpose of our work is to build a model that can handle general navigation meshes, without limitations on the implementation or cell shape (triangle, quads, or convex polygons). Our solution works with any NavMesh, and path finding over the hierarchy is currently done with A*. However, it is not limited to a specific A* implementation, and thus an alternative path-finding algorithm could also be tested.

This paper extends the original HNA* [3] by presenting two methods to solve the bottleneck of the Start/Goal connection step. The first one is based on doing further pre-computation and storing additional data that can be rapidly queried during simulation time, and the second one is based on exploiting parallelism to compute several connecting paths simultaneously. We then present a parallel implementation for multi-agent path finding based on HNA*, to investigate the extent to which HNA* can offer a performance boost for large crowds. Finally we perform a thorough comparison of our results in terms of memory and performance, and run stress tests to determine the number of paths that can be computed in parallel with our methods to handle multi-agent simulation.

4. The HNA* algorithm

The focus of this paper consists of solving the bottleneck that appears in HNA* when connecting start (S) and goal (G) positions into the high-level abstraction graph. Before explaining the details of our new approach, we would like to remind the reader how HNA* performs path finding (for more information we refer the reader to [3]). 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 (see a simple example in Figure 2). Finding a path in this representation consists

of four steps that can be seen in Algorithm 1 (and are illustrated in Figure 3).

Algorithm 1 Find Path with HNA*

```
1:  procedure FindPathHNA*(S, G, L)
2:      if L = 0 then
3:          path ← FindPathA*(S, G, 0)
4:          return path
5:      n_L^S ← getNode(S, L)
6:      n_L^G ← getNode(G, L)
        ▷ Step 1: Connect S and G at level L:
7:      linkNodeToGraph(L, n_L^S)
8:      linkNodeToGraph(L, n_L^G)
        ▷ Step 2: Find path between S & G nodes at level L:
9:      tempPath ← findPathA*(S, G, L)
        ▷ Step 3: Extract subpaths (intra-edges):
10:     for highNode ∈ tempPath do
11:         path ← path + getIntraEdges(highNode, L − 1)
        ▷ Step 4: Remove temporal nodes:
12:     deleteTempNodes(n_L^S, n_L^G)
13:     return path
```

The bottleneck appears in step 1, since it is necessary to compute A* from S to each *inter-edge* in the high-level node, n_L^S. In the original algorithm, these local paths are computed sequentially, first for S and then for G. Therefore, the total cost of Step 1 in HNA* increases rapidly with the number of *inter-edges*. Also 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. 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 (see section 4.1 for the theoretical analysis on the number of *inter-edges* per node.). Figure 4 shows an example where the number of *inter-edges* for a high level node is so large that connecting S and G would be more computationally expensive than simply running A* between S and G at $L0$.

4.1. Theoretical upper bound on the number of inter-edges

Each node n_L^i in level L is created by merging μ nodes of level $L-1$. For the first level, $L1$, of the HNG, n_1^i is created by merging μ nodes of $L0$, which are the polygons in the navigation mesh. Each polygon has s sides, so we can work with triangular meshes when $s = 3$, but also with convex polygons of 4 or more sides. Each *side* of a polygon can either be a *portal* (edge between two walkable polygons), or an *obstacle edge* (edge with an adjacent obstacle or the limits of the map).

When μ polygons are merged to form a node n_1^i (node i in Level 1 of the HNG), then some *portals* will be internal to n_1^i (portals between two polygons that belong to the same high level node, n_1^i), while others will connect a polygon in n_1^i with a polygon in n_1^j (with $i \neq j$). Those portals connecting different nodes in L1 will become *inter-edges* of n_1^i. For the purpose of obtaining an upper bound on the number of *inter-edges* per node, we will consider all polygon edges to be *portals* (this would only be possible if we had a navigation mesh with all polygons having s adjacent walkable cells, but in reality some of those edges will be adjacent to obstacles and thus cannot turn into *inter-edges*).

To compute the upper bound number of *inter-edges* for L1, $I_{(1,\mu)}$, we need to consider the following facts: (1) Merging μ polygons of s sides each to generate n_1^i, means that we have a total of $s \cdot \mu$ edges. (2) Only edges that are not interior to n_1^i can become *inter-edges*, therefore we need to remove those edges that were used for the merging. Merging μ polygons removes at least $2(\mu-1)$ edges (one for each polygon being merged and assuming only one shared side). Note that for $\mu > 2$ there could be even more, but since we are computing the upper bound of the number of *inter-edges* we will be conservative and assume the minimum possible number of removable edges. Therefore we obtain that the number of *inter-edges* for a node in L1 can be computed with equation 1.

$$\begin{aligned} I_{(1,\mu)} &= s\mu - 2(\mu-1) \\ &= \mu(s-2) + 2 \end{aligned} \tag{1}$$

This equation shows that the number of *inter-edges* for L1 increases linearly with the value of μ, as we had observed empirically in our previous work [3].

Similarly, we can compute the number of *inter-edges* for a node n_x^i in level x of the hierarchy Lx, by merging μ nodes n_{x-1}^i. Following the same logic, *inter-edges* of level $(x-1)$ will become *inter-edges* of level x if they belong to the border between two different nodes at level x.

$$I_{(x,\mu)} = \mu I_{(x-1,\mu)} - 2(\mu-1) \tag{2}$$

Which can be written as (proof in Appendix B):

$$I_{(x,\mu)} = \mu^x(s-2) + 2 \tag{3}$$

Equation 3 shows the upper bound for the number of *inter-edges* in a node at level x. Therefore, as we build higher levels in the hierarchy, the number of *inter-edges* increases exponentially. This trend was already observed through experimental analysis in [3].

It is important to note that the number of *portals* will be smaller than s since some of the polygon sides will be obstacles. Moreover, the MLkP method merges nodes minimizing the total number of *inter-edges*. Therefore, in practice this upper bound is never reached. However, it proves the impact on the number of $inter-edges$ per node at level x, with respect to μ and x. In the original HNA* algorithm, connecting S and G with the $n+m$ *inter-edges* (n for S and m for G) was done as $n+m$ sequential calls to the A* algorithm. Thus the cost of such step could become prohibitive for certain configurations. With the two alternative approaches presented in this paper, we are drastically dropping that cost, by either using additional storage or performing the S/G connection step as $n+m$ parallel computations of A*.

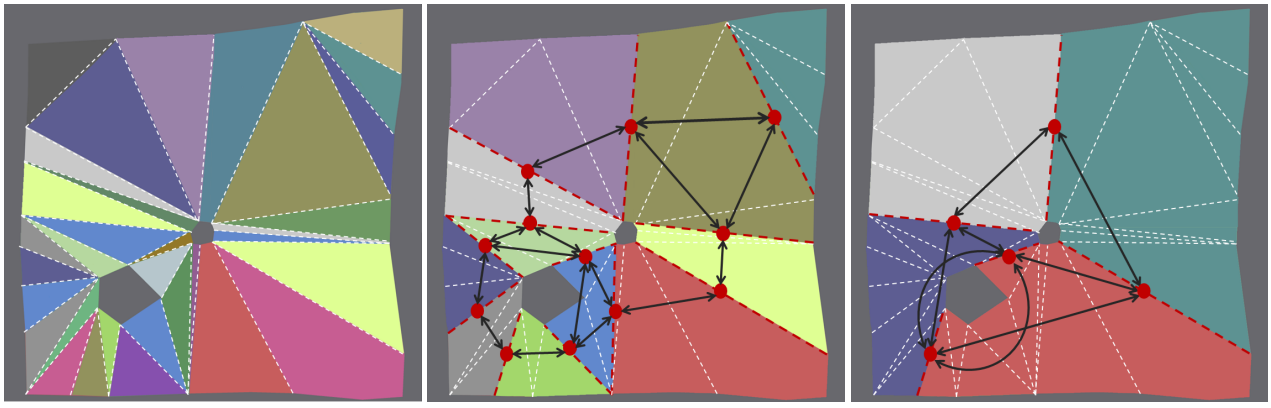


Fig. 2. From left to right we can see a simple map at L0, L1 and L2. The HNG has been built with $\mu = 3$. Note that colors are used to identify nodes in each level, and the overlapping of a node in L1 with colored nodes in L0 visually identifies which nodes in L0 are merged to form a node in L1 and similarly between L1 and L2. White dotted lines indicate portals at L0, red dotted lines in L1 indicate *inter-edges* (connections between nodes at L1), and the same applies for L2 (on the right hand side). Finally, black arrows in L1 and L2 indicate *intra-edges* (pre-computed A* paths to cross a high level node from one *inter-edge* to another). HNG consists of the set of vertices represented by red dots (one per each *inter-edge*), and the set of edges represented by black arrows (one per each *intra-edge*)

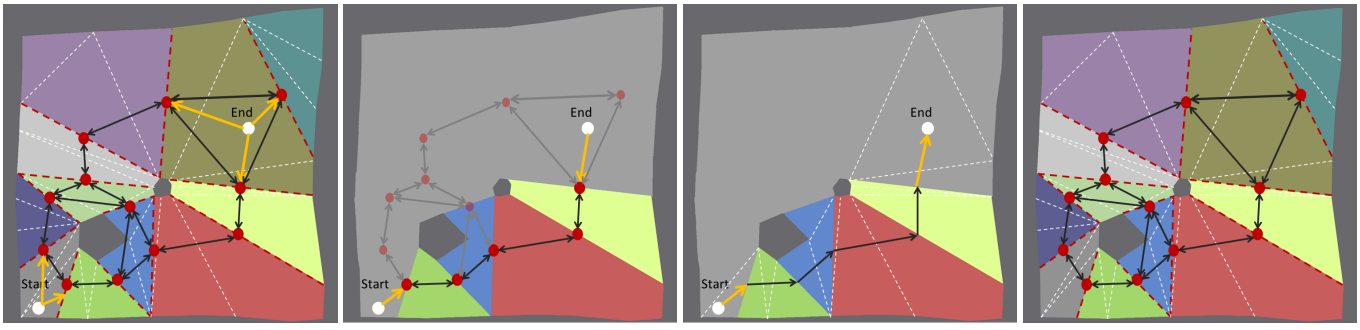


Fig. 3. From left to right we can see the 4 steps of HNA* at L1. Step 1 connects S and G to the HNG by creating temporal connections between S/G and the *inter-edges* of the high level node (yellow arrows). Step 2 computes A* at the HNG (highlights the resulting path). Step 3 extracts the *intra-edges* which contains the sequence of polygons from $L0$. Step 4 removes S/G and the temporal connections to recover the original HNG at L1.

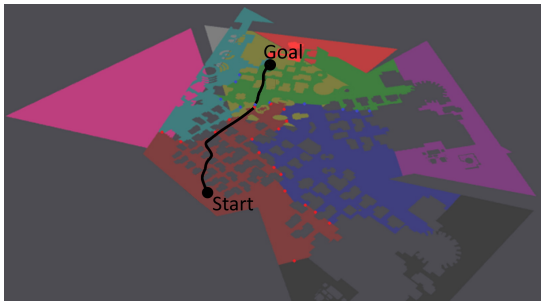


Fig. 4. Example scenario, where connecting S and G becomes a bottleneck due to the large number of *inter-edges*.

5. New Connect S and G approaches

5.1. Pre-Calculated Connecting Paths (PCCP)

The simplest way to solve this problem consists of pre-storing further information to speed-up the connection step. We can calculate the A* path from a 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 (in $L\#$ where # represents the number corresponding to the highest level in the hierarchy). 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 using the A* algorithm and stored in memory using a MultiMap hash table. Table 1 shows an example of such a table. This table has for each cell, one entry per *inter-edge*, with the *Path* and *Cost* information returned by A*. The *Path* stores the sequence of polygon IDs at L0 that the agents will have to walk through to go from that cell to the corresponding *inter-edge*. *Cost* indicates the length of the path from the center of the cell to the *inter-edge*. *Connects to* indicates the high level node reachable with that path. In this particular example, there are 3 *inter-edges* for polygon 18, and thus for entry polyID=18 we can find 3 alternative paths with their corresponding cost. Note that one of the entries for polygon 17 does not show any path for one of the *inter-edges*, because one of its segments is already an inter-edge. These paths would be the temporal connecting edges with the high-

level graph (HNG) in order to compute A* at the higher level of the hierarchy during the on-line phase of the algorithm.

Table 1. Structure of MultiMap for some example nodes in Figure 5.

Polygon	Path	Cost	connects to
p_{10}	13-16	12.5	n_1^1
p_{10}	12-11-14	16.3	n_1^3
p_{17}		2.7	n_1^2
p_{17}	22-23-21-19	22.8	n_1^5
p_{17}	22-23-21-19	21.3	n_1^5
p_{18}	20-15	15.3	n_1^2
p_{18}	20-29	11.4	n_1^4
p_{18}	24	4.9	n_1^1

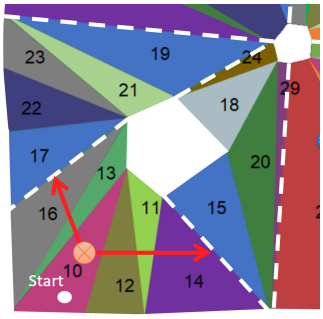


Fig. 5. Section of the example map for L1 with $\mu = 6$. On the left, map at $L0$ with numbers indicating polygon IDs. On the right, L1 of the HNG with numbers indicating node IDs at L1.

Algorithm 2 shows the off-line phase of our method. 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 also to note that center points are computed simply to obtain estimated distances to *inter-edges*, since the real S/G points could lie anywhere in the polygon. However this does not imply that the local movement of the agent has to cross the center point. Agents are simply steered towards the portal connecting to the next cell in their paths, without necessarily going through the center. Since our navigation mesh guarantees that all cells at L0 are convex, then paths are free of collisions against the static geometry and collisions against other moving agents can be handled through steering techniques [1].

During a path search, HNA* needs to find the node containing S and connect it to the high-level graph. The algorithm checks the ID of the polygon containing S, obtains its center position and extracts the temporal edges from the MultiMap table containing PCCP. We thus simplify the connect step with a query for the stored paths of nodes S and G as opposed to computing A* sequentially for each *inter-edge* of the node $n_\#^S$ (node of level # containing S) and $n_\#^G$.

5.2. Parallel search on GPU (CUDA HNA*)

Algorithm 2 Calculate Connecting Path

```
1:  procedure CALCULATE_CONNECTING_PATH()
2:      N ← NumOfPolygons                          ▷ in L0
3:      C ← NumOfCluster                           ▷ in L1
4:      for i ∈ [1..N] do
5:          SId ← GetPolygonID[i]
6:          S ← GetPolygonCenterPos[i]
7:          for k ∈ [1..C] do
8:              CId ← InterEdgeID[k]
9:              if SId = CId then
10:                 G ← InterEdgePos[k]
11:                 {PolyId, Path, Cost} ← Astar(S, G)
12:                 SavePCCP(PolyId, Path, Cost)
13:             end If
14:         end for
15:     end for
```

The problem of connecting S and G to the high-level graph can be run in parallel by computing simultaneously all connecting paths between S/G and the corresponding *inter-edges*. It is possible to use parallelism on the CPU or the GPU, but as shown in Rahmani et al [34] the GPU offers the best performance boost. Therefore in this paper we will focus exclusively on a GPU implementation using CUDA [35].

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. For example, A* algorithm running on the CPU performs many accesses to global memory for storing and retrieving nodes from/to both open and closed lists (implemented as a binary heap), which can turn into a bottleneck. However, the GPU can read and write from local memory much faster than the CPU does with global memory[36]. Therefore, in this work we have used the GPU shared memory facility (local memory). 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), and it 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 [37]. One of the most important factors that can have an effect on performance is the degree of parallelism (DOP), which in our case corresponds to the total number of *inter-edges* (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. Therefore we can now replace the HNA* step 1 by a single kernel call that will run $linkNodeToGraph(L, n_L^S)$ using n parallel threads for each portal in n_L^S, and in parallel it runs $linkNodeToGraph(L, n_L^G)$ using m parallel threads for each portal in n_L^G.

6. Multi-Agent Path Finding

In order to extend our system to handle large crowd simulation, it is necessary to run multiple path searches simultaneously. Having agents compute paths in parallel is an obvious way to speed-up path finding. Therefore one could simply use the basic A* algorithm, but have as many agents computing paths in parallel as the computer architecture allows. The interesting problem here is to determine whether the performance boost of our hierarchical path finding algorithms would also benefit a parallel multi-agent simulation.

This problem is highly parallelizable even with the hierarchical map representations, as we simply need all agents to have access to the hierarchy information. This could be done by either storing it in shared memory or keeping local copies for each agent. The trade-offs to explore are the access to share memory by multiple entities, and the options for local memory based on size and access speed. Note also that the connecting paths for both S and G steps of HNA* are completely independent from each other, so we can still parallelize this step for all agents.

Considering the architecture of 1D CUDA grids and blocks, we have dedicated N blocks (where N is the number of agents) and four threads per each block. The purpose of those four threads is to handle the following steps of the HNA* algorithm (see section 4):

- **Thread 1:** Get the connecting edges for the Start position S (step 1 of HNA*).
- **Thread 2:** Get the connecting edges for the Goal position G (step 1 of HNA*).
- **Thread 3:** Handle synchronization tasks.
- **Thread 4:** Computes the high-level path, extracts *intra-edges* and deletes S and G from the hierarchical navigation graph (HNG). Steps 2, 3 and 4 of HNA* respectively.

The maximum number of agents computing paths in parallel is limited by the number of blocks that CUDA can run in parallel, which is 65,535. Inside each block, we have 4 threads running. Note that thread 4 cannot start until threads 1 and 2 have finished connecting S and G to the inter-edges in the corresponding high level node.

Since the data of the hierarchical navigation graph (HNG), consisting of *intra-edges* (for step 3 of HNA*) needs to be precomputed and stored, we have tested data storage in both texture and local memory to evaluate which one allows faster access. Texture memory shows a friendly cache behavior when we perform several reads that are spatially close to each other [38]. Unlike traditional CPU caches that cache sequential addresses, the GPU texture memory is optimized for 2D spatial locality. In our experimental results we observed that access to texture memory was 1.2x faster than local memory, therefore we decided to use texture memory for our implementation.

The details of our parallel multi-agent path finding method are shown in algorithm 3. The next issue to study is the performance benefits of implementing $LinkNodeToGraph(L, polyId)$ using hash tables (section 6.1) or computing the connecting paths in parallel (section 6.2).

Algorithm 3 Multi-Agent HNA*

```
1:  procedure MultiAgentHNA*(S,G,L)
2:      AgentId ← blockIdx.x;
3:      if L = 0 then
4:          path ← FindPathA(S, G, 0)
5:          return (AgentId, path)
6:      end if
    ▷ Step1. Connect S and G in parallel at level L:
7:      if (threadIdx.x = 1) then
8:          S_polyId = AgentsData[AgentId][0]
9:          n_L^S ← getNode(S, L)
10:         LinkNodeToGraph(L, S_polyId, n_L^S)
11:     end if
12:     if (threadIdx.x = 2) then
13:         G_polyId = AgentsData[AgentId][1]
14:         n_L^G ← getNode(G, L)
15:         LinkNodeToGraph(L, G_polyId, )
16:     end if
17:     if (threadIdx.x = 3) then
18:         syncthreadsCUDA()
19:     end if
20:     if (threadIdx.x = 4) then
    ▷ Step2. Find path between S & G notes at level L:
21:         tempPath ← findPathA(S, G, L)
    ▷ Step3. Extract subpaths:
22:         for highNode ∈ tempPath do
23:             path ← path + getIntraEdges(highNode)
24:         end for
    ▷ Step4. Remove temporal nodes:
25:         deleteTempNode(n_L^S, n_L^G)
26:     end if
27:     return (AgentId, path)
```

6.1. Parallel path finding with PCCP

As described in section 5.1, PCCP uses a hash table to store connecting paths from the center of each polygon to the set of *inter-edges* in the corresponding high level node.

During the online phase, all agents are run in parallel. Each agent computes its own high-level path, querying for both *intra-edges* and connecting paths for S and G from texture memory. Therefore the *LinkGodeToGraph* method in lines 10 and 15 of algorithm 3, is performed with a query to texture memory where all connecting paths were previously stored.

6.2. Parallel path finding with CUDA HPA*

The previous algorithm still required that some portion of the allocated memory for HNA* algorithm was used to store the PCCP information. Since for very large scenarios, the size of allocated memory could become a bottleneck, and computing multiple paths simultaneously is highly parallelizable, we also propose a solution that does not require additional memory other than the hierarchical graph and *intra-edges*. This solution not only saves memory, but it also provides a more scalable solution.

For the parallel implementation of the step connecting S and G, two threads are dedicated to launch two child kernels (one to connect S and another one to connect G).

In order to launch the child kernels to connect S in parallel, we consider m 1D blocks in the 1D CUDA grid where, m is the number of *inter-edges* of the polygon containing S. For each block B_i ($i \in [0, m]$) we compute A* from S to *inter-edge* ie_i (and similarly for G). All connecting paths to S and G are stored in shared memory as temporal edges of the HNG before computing the high-level path.

So the connect step for S and G will take as long as the longest of the A* searches to link S or G to an *inter-edge*. Note that this A* search is computed over a small section of the navigation graph. For example, for the case of connect S, this section corresponds to a set of connected polygons $\langle p_i, p_{i+1}, ...p\mu\rangle$, such that $\forall p_j \in n_x^s$, and $S \in n_x^s$. The number of polygons being limited by the user input value μ, and the hierarchy level Lx. The total number of connects S/G running in parallel is thus:

$$ParallelConnectsSG = \sum_{i=0}^{N}(m(i) + n(i)) \qquad (4)$$

The maximum value of N that can run in parallel is 65,535, and also the maximum number for each agent of $(n + m) \leq 65,535$

So, in this new method, the call *LinkNodeToGraph* (lines 10 and 15 of algorithm 3), to connect S and G to the *inter-edges* of n_L^S and n_L^G respectively, is performed by a kernel path finding search (A*), running all in parallel inside each block.

7. Experimental Results

7.1. Results of PCCP

7.1.1. Error and Memory Usage in PCCP

In this section we present the results achieved in terms of performance, but also discuss the limitations of each approach. All

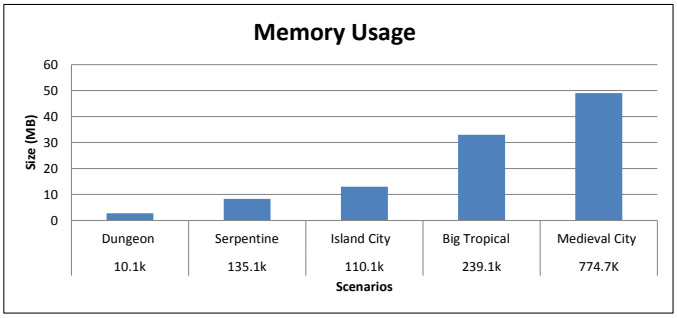


Fig. 6. Memory usage in 5 different size scenarios.

methods described in this paper have been implemented using C++ and CUDA, with an Inter Core i7 Cpu @3.5 Gz, 1 MB L2 cache and 8MB L3 cache, 16 GB RAM. We have used an Nvidia GTX 420 with 2.4GB off-chip global memory and 2496 CUDA core.

As we expected, the pre-calculated paths method (PCCP) achieves the best performance. 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 6 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 6). The allocated memory for the Dungeon scenario with 119 polygon is 2.9 MB while the allocated memory for the Medieval City scenario with 16,867 polygons is 49.6 MB. Memory could be further reduced by storing only the next cell as opposed to the whole path in the hash table. However, this would require further accesses to the hash table, thus reducing performance. In any case, memory usage in PCCP is insignificant for our tested scenarios, so these results confirm that PCCP can be a simple yet powerful way of eliminating the bottleneck of connecting S and G in the original HNA* algorithm. Also note that this information depends on the number of *inter-edges* in each cell, and it is meant to be used for as many path searches as needed. Therefore, memory size is independent of the number of path searches being computed.

In PCCP we have computed paths and costs from the center of each polygon to the *inter-edges* of its cell and stored them in a hash table. When inserting new S and G points in any location of a polygon, the algorithm queries the hash table for paths with the IDs that correspond to the polygons containing S and G. Undoubtedly this introduces an offset between the center positions and the real S and G positions. However, this offset represents only a marginal error when compared to the total length of the path (in most cases, it simply adds a small offset at the beginning and at the end of the total path). The reason is that S and G will not necessarily be located at the center of the cell, and yet the HNA* search is done assuming that they are. However, it is also important to emphasize 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. Figure 7 shows the

difference in path length between the proposed pre-calculated path method (PCCP) and the original HNA* method. For this figure, 100 random S/G position were used to compute paths with PCCP and HNA* in the Medieval City scenario shown in Figure 8c. The horizontal axis shows the 100 paths sorted in ascending order based on the distance between S and G. Our experimental results show a small impact on the total length of the path (3% on average for paths over 100m, and 5% on average for shorter paths).

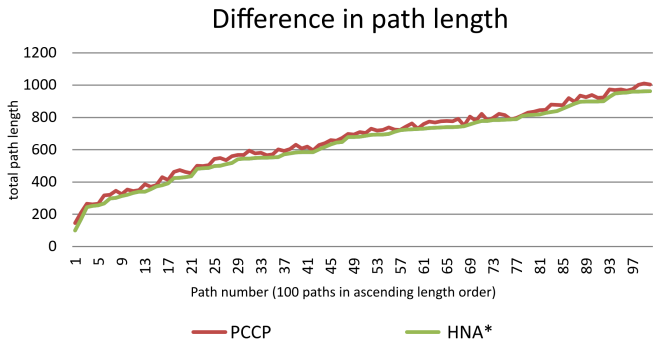


Fig. 7. Difference in total path length between PCCP HNA* and the original HNA*.

Table 2. Number of triangles in the environments and number of polygons in the navigation mesh, which corresponds to the graph size at level 0 in the HNG.

Map Name	Geometry # Triangles	NavMesh # Poly
Serpentin City	135.1K	3.9K
City Island	110.3K	5.5K
Medieval City	774.7K	16.9K
Tropical Island	239.1K	12.7K

7.1.2. Performance Results for PCCP (1 agent)

For the evaluation of this method we have used several 3D scenarios as shown in Figure 8, 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. Paths are computed for up to 3 levels in the hierarchy and increasing values of μ = {2,4,6,8,10,15,20}, where μ indicates the number of nodes merged from one level to the next one of the hierarchy.

For the City Island scenario, we can see in Figure 9-a1 the average cost of performing A* in this scenario is 2.2ms (Note that A* is always computed on the navigation graph at L0, and thus it is not affected by the hierarchy configuration). Figure 9-a1 shows that the performance of the PCCP 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 9-a2 . The strength of the pre-calculated paths (PCCP) can be observed for higher levels of the hierarchy. Figure 9-b1 and Figure 9-c1 show

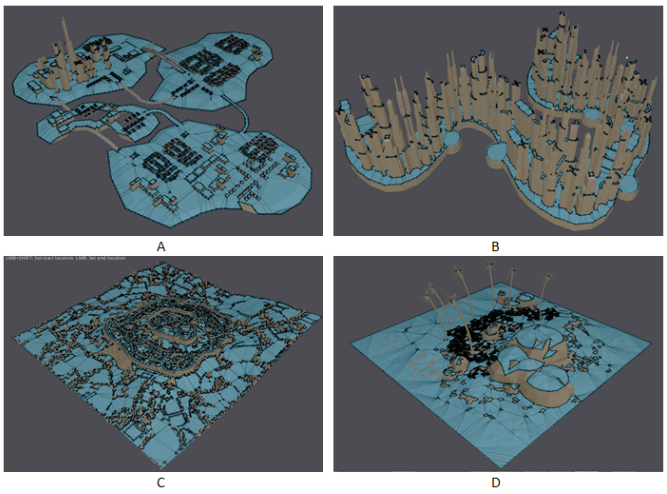


Fig. 8. Different scenarios with their corresponding number of triangles in the mesh. A: City Island (110.3K), B: Serpentine City (135.1K), C: Medieval City (774.7K) and D: Big Tropical scenario (239.1K).

significant performance improvements when compared against HNA*. These improvements can be seen in Figure 9-b2 and Figure 9-c2 where we have clearly managed to drop the cost of the connecting S and G step. Compared to A*, PCCP provides its largest speedup (9.3x faster) for L = 2 and μ = 10.

Results are similar for the Big Tropical Island (Figure 13). The average cost of performing A* in this scenario is 1.7ms. 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. The advantages of the new implementation are noticeable for L2 and L3 after a specific value of μ. HNA* had a performance of 2.06ms for L2 and μ = 20 and 9.9ms for L3 and μ = 10 while PCCP HNA* obtained paths in 0.39ms for L2 and μ=20 (4.3x faster than A*), and 0.25ms for L3 and μ = 10 (6.8x faster than A*).

Similar results were obtained for the Medieval city scenario (A* performance of 3ms) (Figure 14). HNA* suffered from the insert S/G bottleneck after a specific value of μ. With 1.28ms in L2 and 3.29ms in L3 for μ = 15 while PCCP HNA* had a computational time of 1.83ms (1.6x faster than A*) in L2 for μ = 20 and 1.76ms (1.7x faster than A*) in L3 for μ = 10, thus offering a speed-up for all configurations.

7.2. Achieved Results of Parallel Search on the GPU (1 agent)

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*. As we described earlier, 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 10 for the example of $L2$.

To compare the CUDA method against PCCP and HNA*, we have computed the cost of calculating 100 paths in the same scenarios and configuration. The GPU was a single NVIDIA

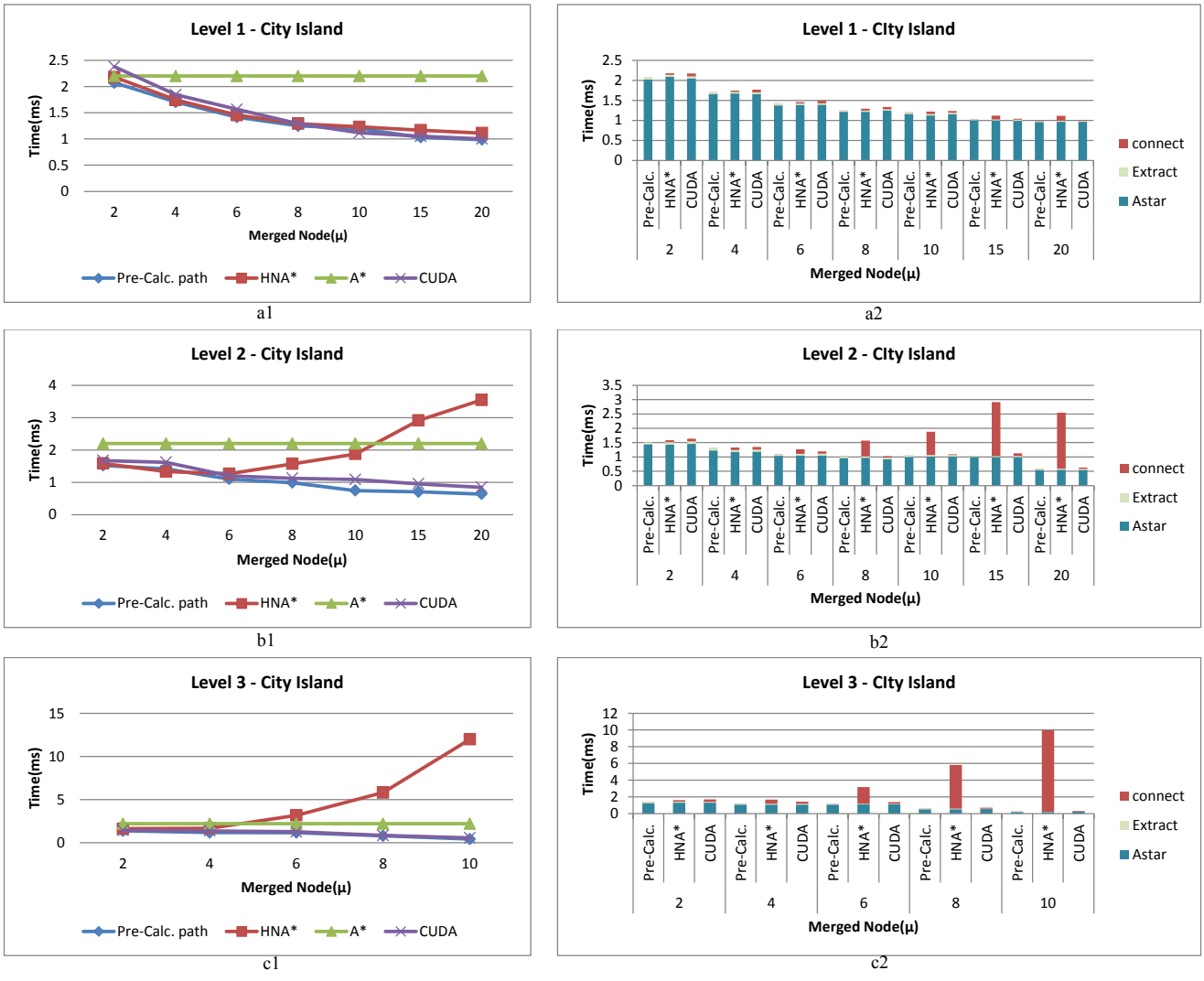


Fig. 9. Performance results for the city Island scenario.

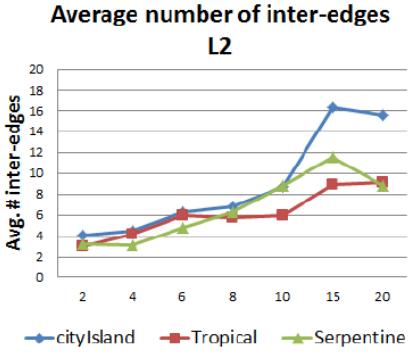


Fig. 10. Average number of *inter-edges* for $L2$ as the value of μ increases.

Geforce GTX 420 with 2.4GB off-chip global memory and 2496 CUDA cores.

For the City Island scenario, Figure 9-a1 shows that the average cost of performing A* in this scenario is 2.2 ms. Figure 9-a1, for $L1$ of hierarchy and $\mu = [2, 20]$ the performance of PCCP is better 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 $0.628ms$, and for $L3$ and $\mu = 10$ is down to $0.33ms$ for CUDA (6.7x faster than A*).

As we can see in the right column of Figure 9, 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 (Figure 13), the performance differences are not relevant for L1, but show drastic improvements from L2 onwards. For instance, the computational time of CUDA in L2 and for μ=20 drops to 0.404ms (4.2x faster than A*) while the computational time of HNA* increases up to $2.06ms$. However, Pre-Calculated paths (PCCP) is still faster than CUDA in L2 with the time being $0.396ms$ for

Table 3. Maximum number of agents that can run in real time (25FPS) in sequential multi-agent path finding for each algorithm.

Map Name & hierarchy configuration	A*	HNA*		
		Original	CUDA	PCCP
City Island ($L1\mu20$)	18	11	48	63
Tropical Island ($L2\mu10$)	24	5	62	140

μ=20.

Finally, we obtain similar results for the Medieval city scenario (Figure 14). In the original HNA* algorithm, the time of connecting S and G points increases up to 9.05*ms* in L3 for μ =10 whilst it drops to 1.88*ms* for CUDA (1.6x faster than A*), and 1.76*ms* for Pre-calculated path.

7.3. Multi-agent parallel path finding

If we performed path finding for multi-agent systems in a sequential manner, we would have strong limitations on how many agents we could run in real time. However, the exact number of agents depends strongly on the map and the hierarchy configuration (especially for the old HNA*). For example, if we consider that we run sequential path-finding based on A* and HNA* (for the original, CUDA and PCCP approaches), we would obtain that the maximum number of agents that could be run on average are those shown in table 3.

Therefore, if we would like to compute path finding for a large number of agents, it is necessary to apply parallelism also at the level of each agent's computation.

We have evaluated the performance of parallel multi-agent path finding, using the two methods described in this paper (PCCP and CUDA-HNA*), and A*. All three methods use the same CUDA implementation to compute all agent's path in parallel for a multi-agent system. This will allow us to study, whether the gain that we can achieve with a hierarchical path finder for a single agent, also holds when using multi-agent parallel path finding. For this comparison, we have used again the 4 scenarios shown in Figure 8 and compared three algorithms:(1) A*, (2) PCCP, and (3) CUDA-HNA*.

As shown in Figure 11, performance times increase in all four methods with the number of agents. Performance of the multi-agent parallel PCCP method is the fastest, followed by the CUDA-HNA* version, which also outperforms A*. As we can see, the parallel implementation in CUDA can handle real time path finding for over 500K agents even when using the basic A* algorithm, but with an important speed-up achieved by using HNA* with the connection step in parallel. We have chosen the number of agents for our simulation to show that the jumps in the computational time are due to the number of blocks available to run 65,535 agents in parallel. With our CUDA version, we had 65,535 number of blocks to launch our parallel path finding, which is consistent with the computational jumps appearing for each multiple of 65,535 agents. The negative impact of the number of blocks is much more noticeable for A*, than for our implementations with PCCP or CUDA, making our HNA* more scalable.

Our main interest with this thorough evaluation was to determine whether HNA* offered important speedups for multi-agent parallel implementation. As we can see in Figure 12, for the 4 scenarios tested, we can observe speedups on average between 4.3x and 15.7x for PCCP, and between 3.6x and 9.8x for CUDA-HNA*. Therefore, the benefits of our hierarchical representations still hold even when a parallel implementation could be carried out for both HNA* and A*.

8. Conclusion

In this paper we have studied the problems of path finding in large scenarios for hierarchical representations based on navigation meshes. We have presented two solutions to the S/G connection step. The first one consists of using pre-calculated paths (PCCP) 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 online search. The second one takes advantage of the highly parallel nature of the problem, and presents a new approach using CUDA, so that all sub-paths to connect S and G can be computed in parallel. To evaluate our different methods we have used several 3D scenarios with increasing numbers of cells in their navigation mesh and increasing numbers of merged polygons. Our results show that both PCCP and CUDA methods are much faster than the original HNA*. PCCP requires more memory usage than CUDA, although this does not present a limitation. For all tested scenarios, the performance improvements are not very important for L1, but they become substantial from L2 onwards, as they eliminate the bottleneck of HNA* which was the connect S and G step. With these improvements, we have completely eliminated the important bottleneck from HNA* and thus obtain a hierarchical path finding algorithm for general navigation meshes that offers great speed-ups for a larger number of scenarios, regardless of the hierarchy configuration.

Finally we have carried out a thorough performance comparison of a parallel multi-agent implementation of A*, PCCP and CUDA-HNA* in order to determine the potential of using hierarchical path finding. For this comparison all three methods can compute path finding for multiple agents in parallel. As we have shown in our results, the speed-ups achieved by both our methods outperform the parallel A* solution. Therefore hierarchical implementations can allow us to run a potentially much larger number of agents simultaneously.

Acknowledgments

This work has been funded by the Spanish Ministry of Science and Innovation and FEDER under grant TIN2017-88515-C2-1-R.

References

[1] Pelechano, N, Allbeck, JM, Badler, NI. Controlling individual agents in high-density crowd simulation. In: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation. Eurographics Association; 2007, p. 99–108.

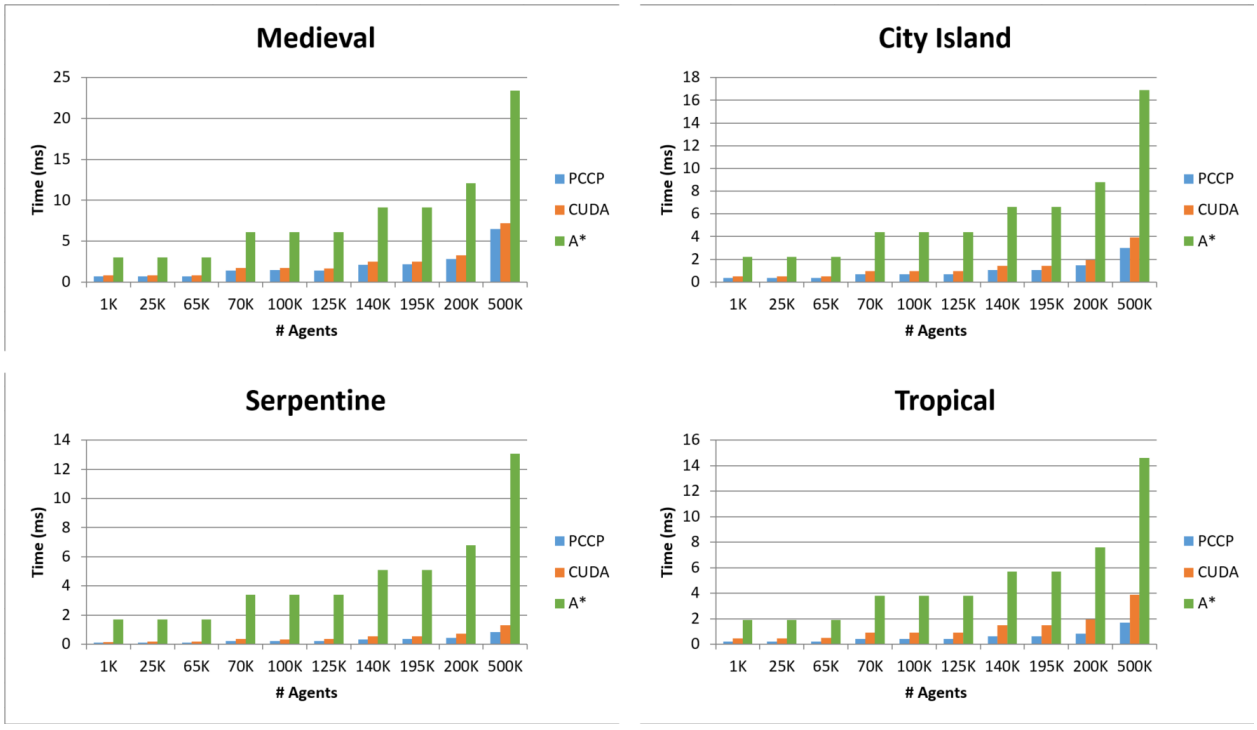


Fig. 11. Time taken in *ms* to compute the corresponding number of agent's paths in parallel for A*, PCCP and CUDA-HNA. From 1K to 500K agents computing paths simultaneously under 6.5*ms* for PCCP and 7.2*ms* for CUDA.

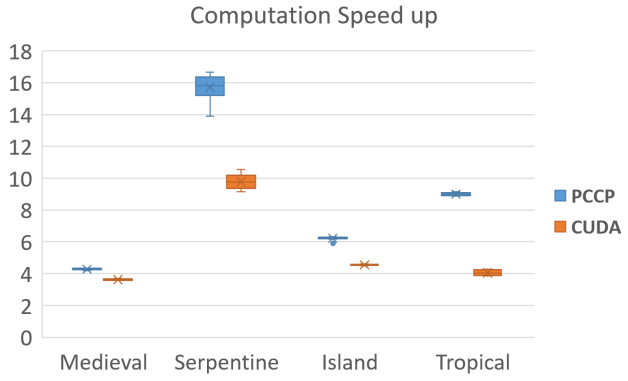


Fig. 12. Speed-up achieved for each of the scenarios, with PCCP and CUDA-HNA* over A*.

[2] Botea, A, Müller, M, Schaeffer, J. Near optimal hierarchical path-finding. Journal of game development 2004;1(1):7–28.

[3] Pelechano, N, Fuentes, C. Hierarchical path-finding for navigation meshes (hna*). Computers & Graphics 2016;59:68–78.

[4] Sturtevant, NR. Memory-efficient abstractions for pathfinding. AIIDE 2007;684:31–36.

[5] Kallmann, M, Kapadia, M. Geometric and Discrete Path Planning for Interactive Virtual Worlds. Morgan and Claypool; 2016.

[6] Van Toll, W, Triesscheijn, R, Kallmann, M, Oliva, R, Pelechano, N, Pettré, J, et al. A comparative study of navigation meshes. In: Proceedings of the 9th International Conference on Motion in Games. ACM; 2016, p. 91–100.

[7] Recast 2017;`https://github.com/recastnavigation/recastnavigation`.

[8] Hart, PE, Nilsson, NJ, Raphael, B. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics 1968;4(2):100–107.

[9] Sacerdoti, ED. Planning in a hierarchy of abstraction spaces. Artificial Intelligence 1974;5(2):115 – 135.

[10] Holte, R, Holte, RC, Perez, M, Perez, MB, Zimmer, RM, Zimmer, RM, et al. Hierarchical a*: Searching abstraction hierarchies efficiently. In: In Proceedings of the National Conference on Artificial Intelligence. 1996, p. 530–535.

[11] Holte, RC, Mkadmi, T, Zimmer, RM, MacDonald, AJ. Speeding up problem solving by abstraction: A graph oriented approach. Artificial Intelligence 1996;85(1):321–361.

[12] Sturtevant, N, Jansen, R. An analysis of map-based abstraction and refinement. In: Miguel, I, Ruml, W, editors. Abstraction, Reformulation, and Approximation; vol. 4612 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. ISBN 978-3-540-73579-3; 2007, p. 344–358.

[13] Rabin, S. A* speed optimizations. Game Programming 2000;:272278.

[14] Bulitko, V, Björnsson, Y, Lawrence, R. Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding. Journal of Artificial Intelligence Research (JAIR) 2010;39:269–300.

[15] Sturtevant, NR, Geisberger, R. A Comparison of High-Level Approaches for Speeding Up Pathfinding. In: Youngblood, GM, Bulitko, V, editors. AIIDE. The AAAI Press; 2010,.

[16] Harabor, D, Botea, A. Hierarchical path planning for multi-size agents in heterogeneous environments. In: Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium on. IEEE; 2008, p. 258–265.

[17] Sturtevant, N, Buro, M. Partial pathfinding using map abstraction and refinement. In: AAAI; vol. 5. 2005, p. 1392–1397.

[18] Lawrence, R, Bulitko, V. Database-driven real-time heuristic search in video-game pathfinding. IEEE Transactions on Computational Intelligence and AI in Games 2012;5(3):227–241.

[19] Kring, AW, Champandard, AJ, Samarin, N. Dhpa* and shpa*: efficient hierarchical pathfinding in dynamic and static game worlds. In: Sixth Artificial Intelligence and Interactive Digital Entertainment Conference. 2010,.

[20] Oliva, R, Pelechano, N. Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments. Computers & Graphics 2013;37(5):403–412.

[21] Mononen, M. Recast navigation toolkit. 2009.

[22] Kapadia, M, Beacco, A, Garcia, F, Reddy, V, Pelechano, N, Badler, NI. Multi-domain real-time planning in dynamic environments. In: Proceed-

ings of the 12th ACM SIGGRAPH/Eurographics symposium on computer animation. ACM; 2013, p. 115–124.
[23] Ammar, A, Bennaceur, H, Châari, I, Koubâa, A, Alajlan, M. Relaxed dijkstra and a* with linear complexity for robot path planning problems in large-scale grid environments. Soft Computing 2016;20(10):4149–4171.
[24] Nash, A, Daniel, K, Koenig, S, Felner, A. Theta^*: Any-angle path planning on grids. In: AAAI. 2007, p. 1177–1183.
[25] Oh, S, Leong, HW. Strict theta*: Shorter motion path planning using taut paths. In: ICAPS. 2016, p. 253–257.
[26] Caggianese, G, Erra, U. Exploiting gpus for multi-agent path planning on grid maps. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). IEEE; 2012, p. 482–488.
[27] Merrill, D, Garland, M, Grimshaw, A. Scalable gpu graph traversal. In: Acm Sigplan Notices; vol. 47. ACM; 2012, p. 117–128.
[28] Ortega-Arranz, H, Torres, Y, Llanos, DR, Gonzalez-Escribano, A. A new gpu-based approach to the shortest path problem. In: 2013 International Conference on High Performance Computing & Simulation (HPCS). IEEE; 2013, p. 505–511.
[29] Caggianese, G, Erra, U. Exploiting gpus for multi-agent path planning on grid maps. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). IEEE; 2019, p. 482–488.
[30] Sigurdson, D, Bulitko, V, Yeoh, W, Hernández, C, Koenig, S. Multi-agent pathfinding with real-time heuristic search. In: 2018 IEEE Conference on Computational Intelligence and Games (CIG). IEEE; 2018, p. 1–8.
[31] Li, J, Surynek, P, Felner, A, Ma, H. Multi-agent path finding for large agents. AAAI; 2019,.
[32] Garcia, FM, Kapadia, M, Badler, NI. Gpu-based dynamic search on adaptive resolution grids. In: 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE; 2014, p. 1631–1638.
[33] Farias, R, Kallmann, M. Optimal path maps on the gpu. IEEE transactions on visualization and computer graphics 2019;.
[34] Rahmani, V, Pelechano, N. Improvements to hierarchical pathfinding for navigation meshes. In: Proceedings of the Tenth International Conference on Motion in Games. ACM; 2017, p. 8.
[35] NVIDIA. CUDA. 2017. `http://www.nvidia.com/object/cuda_home_new.html`.
[36] Zhou, Y, Zeng, J. Massively parallel a* search on a gpu. Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence; 2015,.
[37] Nickolls, J, Buck, I, Garland, M, Skadron, K. Scalable parallel programming with CUDA. Queue 2008;6(2):40–53.
[38] Liu, Y, Zou, Q, Luo, S. Gpu accelerated fourier cross correlation computation and its application in template matching. In: International Conference on High Performance Networking, Computing and Communication Systems. Springer; 2011, p. 484–491.

APPENDIX A

If a path exists over the original navigation mesh, G_0, $P_0(S,G) = \langle p_S, p_1, p_2, ..., p_G \rangle$, then there will also be a path at level Lx. Computing path finding in HNG_x gives as a result the path $P_x(S,G) = \langle \pi^S_{temp}, \pi^{s(dp)}_x, \pi^{p(sq)}_x, ..., \pi^{r((m-1)m)}_x, \pi^G_{temp} \rangle$. $P_x(S,G)$ is the high level path.

Proof. Starting with the sequence of polygons in the path:

$$P_0(S,G) = \langle p_S, p_1, p_2, ..., p_G \rangle \tag{5}$$

the starting polygon p_S will be inside a high level node n^s_x in the HNG_x. Moving from left to right in the sequence of polygons while $p_i \in n^s_x$, we will eventually reach a polygon p_j such that $p_j \in n^d_x$, where $d \neq s$. According to the definition of *inter-edge* given in Section 2, this means that there is an *inter-edge* ι^{sd}_x connecting the nodes n^s_x and n^d_x in HNG_x The sequence of polygons from p_s to p_{j-1} correspond to the temporal path connecting S to the *inter-edge* ι^{sd}_x:

$$\pi^S_{temp} = \langle p_S, ..., p_i, ..., p_{j-1} \rangle \tag{6}$$

From p_j, we can continue sequentially while the polygons we encounter are still inside n^d_x, until there is a polygon p_k such that $p_k \in n^p_x$, and $p \neq d$. By the definition of *inter-edge*, there is an *inter-edge*, ι^{dp}_x, that connects n^d_x, with n^p_x.

Also, the sequence of nodes $\langle p_j, p_{j+1}, ..., p_{k-1} \rangle$ indicates that there is a path traversing the node n^d_x between the *inter-edges* ι^{sd}_x and ι^{dp}_x, which guarantees that there is at least one path between those two *inter-edges*, and thus there will be an *intra-edge*, $\exists \pi^{s(dp)}_x$. Note that this does not mean that the stored *intra-edge* is specifically the sequence $\langle p_j, p_{j+1}, ..., p_{k-1} \rangle$, since the subpath P' was computed with A* between optimal positions inside the polygons in the navigation mesh, whereas $\pi^{s(dp)}_x$ was computed assuming the position at the center of the polygons. If there was only one possible path, then this path will be the optimal, and thus we will have $P' = \pi^{s(dp)}_x$. So our proof guarantees that if there is a path in $P0$ crossing a node, there must be an *intra-edge* to cross the node.

$$\pi^{s(dp)}_x = \langle p_j, p_{j+1}, ..., p_{k-1} \rangle \tag{7}$$

Following the same logic, it is thus straight forward to proof that every sequence of polygons inside the same high level node, guarantees that there will be an *intra-edge* traversing such node, and that for every pair of polygons in the sequence, which appear inside different high level nodes, there will be an *inter-edge* in the HNG_x.

The the last sequence of nodes that are inside the high level node containing p_G, will correspond to the temporal path connecting G π^G_{temp}

$$\pi^G_{temp} = \langle p_{l+1}, p_{l+2}, ..., p_G \rangle, \forall p_i \in n^q_x \tag{8}$$

Finally we can see that replacing the subsequence of polygons from temporal paths and *intra-edges* guarantees that there will have a path between S and G:

$$
\begin{aligned}
P_x(S,G) &= \langle \pi^S_{temp}, \pi^{s(dp)}_x, \pi^{p(sq)}_x, ..., \pi^G_{temp} \rangle \\
&= \langle < p_S, ..., p_{j-1} >, \pi^{s(dp)}_x, \pi^{p(sq)}_x, ..., \pi^G_{temp} \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \pi^{p(sq)}_x, ..., \pi^G_{temp} \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \\
&< p_k, p_{k+1}, ..., p_{m-1} >, ..., \pi^G_{temp} \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \\
&< p_k, p_{k+1}, ..., p_{m-1} >, ..., < p_{l+1}, p_{l+2}, ..., p_G > \rangle \\
&= \langle p_S, p_1, p_2, ..., p_G \rangle \\
&= P_0(S,G)
\end{aligned}
\tag{9}
$$

□

APPENDIX B:

Proof. The number of *inter-edges* increases exponentially with the number of levels in the hierarchy:

As indicated in Equation 10, the number of inter-edges for a node in level x are:

$$
I_{(x,\mu)} = \mu I_{(x-1,\mu)} - 2(\mu - 1) \tag{10}
$$

if we replace $I_{(x-1,\mu)}$ by its corresponding equation, then we have:

$$
\begin{aligned}
I_{(x,\mu)} &= \mu \left[\mu I_{(x-2,\mu)} - 2(\mu - 1) \right] - 2(\mu - 1) \\
&= \mu^2 I_{(x-2,\mu)} - 2\mu(\mu - 1)) - 2\mu + 2) \\
&= \mu^2 (I_{(x-2,\mu)} - 2) + 2
\end{aligned}
\tag{11}
$$

replacing the next level down in the hierarchy, then we have:

$$
\begin{aligned}
I_{(x,\mu)} &= \mu^2 \left[\mu I_{(x-3,\mu)} - 2(\mu - 1) - 2 \right] + 2 \\
&= \mu^2 \left[\mu I_{(x-3,\mu)} - 2\mu + 2 - 2 \right] + 2 \\
&= \mu^3 (I_{(x-3,\mu)} - 2) + 2
\end{aligned}
\tag{12}
$$

and if we continue recursively until we reach level 1:

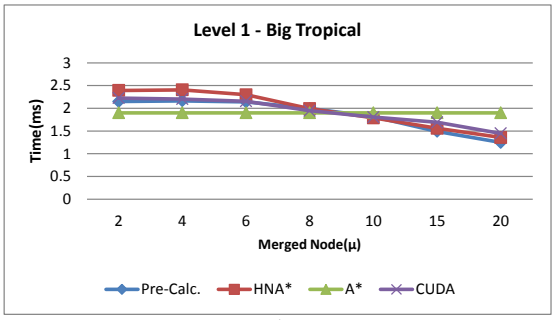
$$
I_{(x,\mu)} = \mu^{x-1} (I_{(1,\mu)} - 2) + 2 \tag{13}
$$

finally, replacing $I_{(1,\mu)}$ by Equation 10 we obtain:

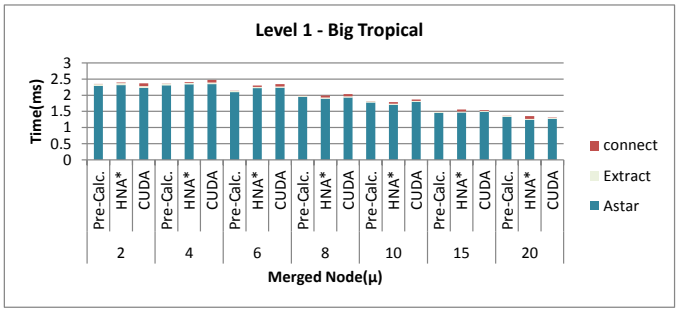
$$
\begin{aligned}
I_{(x,\mu)} &= \mu^{x-1} (\mu(s-2) + 2 - 2) + 2 \\
&= \mu^x (s-2) + 2
\end{aligned}
\tag{14}
$$

APPENDIX C

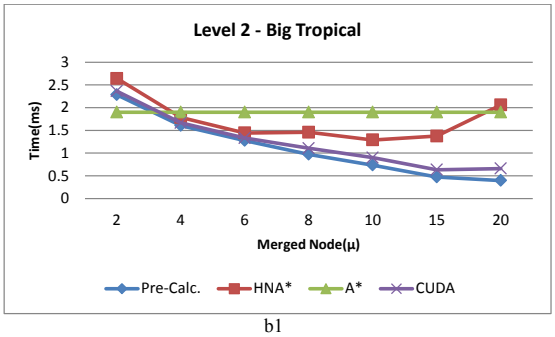
Additional Performance Results for PCCP and CUDA versions of HNA* are shown for two more scenarios: Tropical Island, and Medieval City.



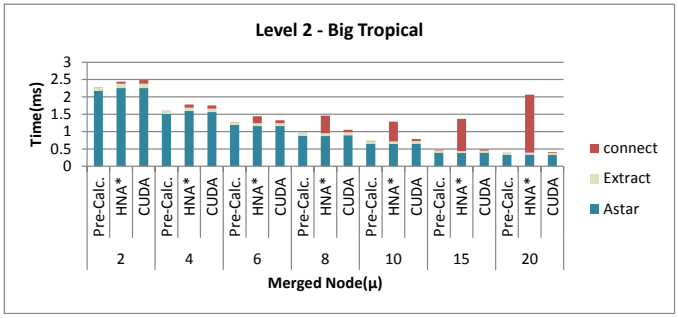
a1



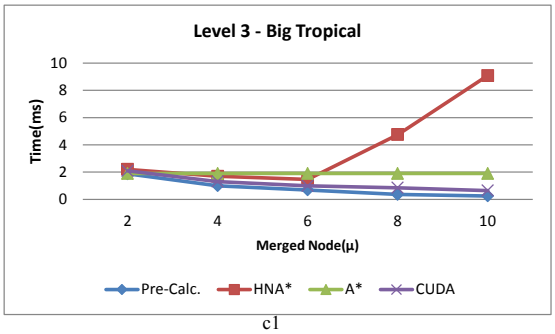
a2



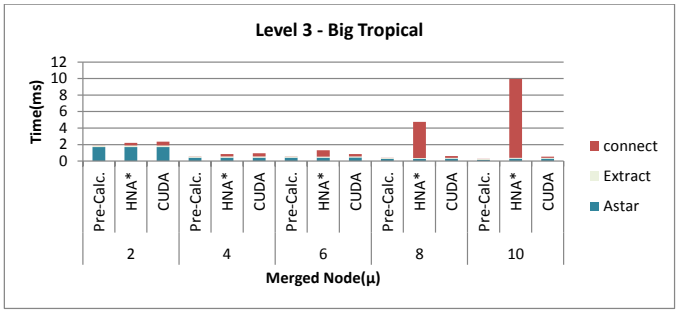
b1



b2



c1



c2

Fig. 13. Performance results for the big tropical scenario.

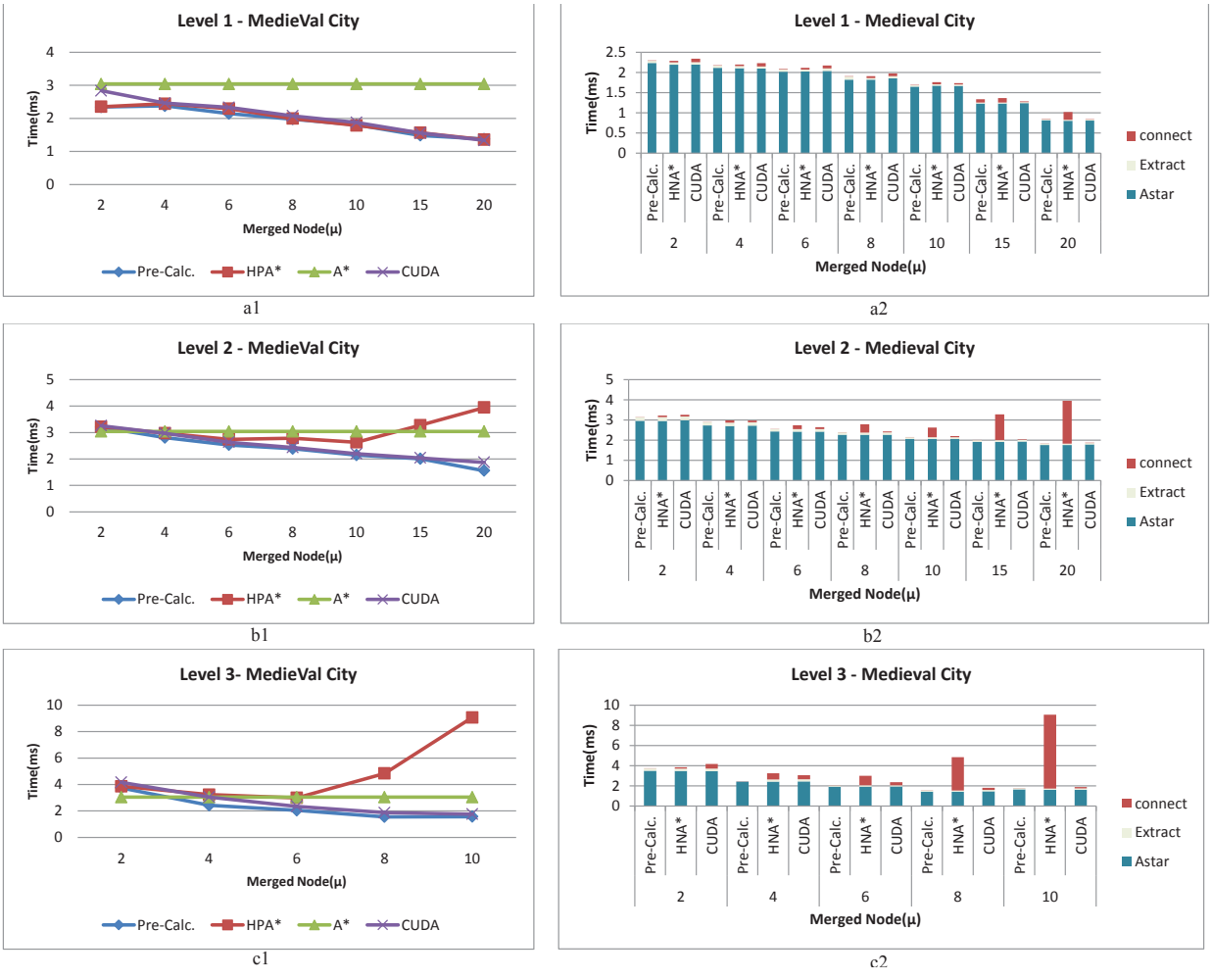


Fig. 14. Performance results for the Medieval city scenario.