COMPUTER ANIMATION AND VIRTUAL WORLDS
Comp. Anim. Virtual Worlds 2012; **23**:73–85
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cav.1427

SPECIAL ISSUE PAPER

Multi-core scalable and efficient pathfinding with Parallel Ripple Search

Sandy Brand and Rafael Bidarra*

Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

ABSTRACT

Game developers are often faced with very demanding requirements on huge numbers of agents moving naturally through increasingly large and detailed virtual worlds. With the advent of multi-core architectures, new approaches to accelerate expensive pathfinding operations are worth being investigated. Traditional single-processor pathfinding strategies, such as A* and its derivatives, have been long praised for their flexibility. We implemented several parallel versions of such algorithms to analyze their intrinsic behavior, concluding that they have a large overhead, yield far from optimal paths, do not scale up to many cores or are cache unfriendly. In this article, we propose *Parallel Ripple Search*, a novel parallel pathfinding algorithm that largely solves these limitations. It utilizes a high-level graph to assign local search areas to CPU cores at "equidistant" intervals. These cores then use A* flooding behavior to expand towards each other, yielding good "guesstimate points" at border touch on. The process does not rely on expensive parallel programming synchronization locks but instead relies on the opportunistic use of node collisions among cooperating cores, exploiting the multi-core's shared memory architecture. As a result, all cores effectively run at full speed until enough way-points are found. We show that this approach is a fast, practical and scalable solution and that it flexibly handles dynamic obstacles in a natural way.

KEYWORDS

Parallel Ripple Search; pathfinding; parallel algorithms; multi-core architectures

***Correspondence**

Rafael Bidarra, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands.
E-mail: R.Bidarra@tudelft.nl

1. INTRODUCTION AND PREVIOUS WORK

As virtual game worlds grow increasingly larger, pathfinding has once again come into the spotlight. The basic motivation for this is that being a computationally expensive but indispensable component in many games, any performance gains here will typically bring about noticeable improvements. In this line, more attention is currently being paid to re-designing pathfinding algorithms, so that they better suit current multi-core architectures.

Classic pathfinding algorithms, such as A* and its many derivatives, have been long praised by game developers for their flexibility and completeness. A* is a best-first search strategy that relies on a cost computing function $f(n) = g(n) + h(n)$ for providing rough cost estimations of a path running through a node n of a search graph [1]. Function $g(n)$ represents the currently known cost for reaching node n from the start node S, and heuristic estimation function $h(n)$ is often implemented by using a cheap "guesstimate" of the remaining travel distance, such as a Manhattan or Euclidean distance, between node n and the goal node G. This heuristic function effectively controls how A* floods its search space. Moreover, $h(n)$ results in optimal paths as long as it remains "admissible;" that is, it never overestimates the true cost for an actual path between node n and the goal node G.

The A* algorithm utilizes the $f(n)$ function to maintain a sorted *Open* list of most promising search candidates while it iterates through the search space, which is also its most computationally expensive component. For each iteration, the algorithm will remove the most promising candidate and place on the list all its not yet visited neighbors. If a neighbor node was already in the *Open* list, A* will perform a crucial "correction step:" it determines if a cheaper path was possible through the candidate node and, if so, modifies its entry accordingly in the *Open* list.

As the flood boundary grows, the algorithm takes increasingly more time to find each successive node that forms the desired path. Empirically, the node that is halfway down the resulting path is closed at roughly a third of the total time taken to find the complete path. This sorting component has also proven a road block for parallelization attempts because it institutes a data dependency

that would generate much communication overhead on distributed processing architectures. Many past attempts have already been made to eliminate this need for sorting on single-processor architectures such as *Iterative Deepening A** (or IDA* for short) [2] but have not resulted in easier distributed processing schemes.

A late addition to the family of A* derivatives, which we used extensively in this research, is called *Fringe Search* (FS); see [3]. FS avoids the need for a sorted candidate list by simply keeping track of all nodes at its search boundary (or "fringe") and opening those that are less expensive than a certain threshold value, which is iteratively incremented. Although this now forces the algorithm to scan through a large unsorted *Now* list from begin to end, it gains its speedup by making the actual visitations extremely cheap. Each node that has an f(n) value higher than the threshold value will simply be moved to a *Later* list. Each new iteration starts by simply swapping the *Now* and *Later* lists, and the node/list manipulations themselves can be implemented very effectively using simple pointer logic. Normally, one will choose the lowest f(n) value in the current *Now* list as the new threshold value. This will make FS behave in the exact same way of a classical A* implementation, which will keep opening the most promising node first, but now without having to explicitly sort node lists before each pass (in the same way that IDA* forgoes this).

In most FS-based pathfinding approaches that we discuss in this article, manipulating the value of this threshold proved to be very useful, because it allowed FS to open multiple nodes per pass. In practice, this means that we sort the node lists "less thoroughly" and instead just open a larger number of most promising nodes all at once. As we will see, although this modification no longer guarantees optimal paths, in practice it significantly increases performance while still yielding paths of a fairly high quality.

Recent path finder parallelization attempts [4,5] have mainly focused on translating A* variants into shaders, so that they can run on graphics processing units or similar vector processors. These schemes benefit either from taking workload off the main CPU or by running pathfinders for a large amount of agents in parallel. Although such approaches have been very successful, these also have the drawback that they take up precious resources that one would rather devote solely to rendering graphics. Also, from a practical point of view, a game-play programmer will have to take extensive measures to apply such pathfinding approaches without serious disruptions of the rendering pipeline. For multi-core architectures, specifically, the world simulation and rendering logic are often running in separate threads that are uniquely assigned to specific cores.

A more traditional approach has been to parallelize A* and related algorithms and make them more suitable for distributed computing on CPU clusters and grids [6–8]. Although these attempts have demonstrated beneficial advantages, they also require more "exotic" hardware and software approaches such as Message Passing Interface that are highly uncommon on virtually any gaming hardware platform up to this date. For practical purposes, these approaches are thus unsuitable and have currently no real relevance in the game development industry.

In conclusion, there is a definite need for simple and portable variants of the A* algorithm that successfully and efficiently exploit today's multi-core architectures. This article first describes and compares a number of parallel pathfinding implementations, focusing on the efficiency of their multi-core use. They all rely on the underlying hardware to implicitly perform the necessary synchronizations, without any blocking. We then introduce a novel algorithm called *Parallel Ripple Search* (PRS) that can easily scale with the number of available CPU cores. It requires no special libraries or hardware interfacing, nor any special synchronization primitives.

2. PARALLEL PATHFINDING IMPLEMENTATIONS

In this section, we describe our investigation on a number of parallelized variants of A* (actually, of FS, for the first two algorithms discussed) to study how they perform on multi-core architectures. Please refer to [9] for a detailed discussion of each algorithm, including its pseudo-code. This study gave us significant insight on how to effectively utilize the computing potential of these architectures for pathfinding purposes.

2.1. Parallel Bidirectional Search

The most obvious strategy to use *two* CPU cores for pathfinding is to have them start at each path extremity, search towards each other and let them "meet halfway;" hence, the name *Parallel Bidirectional Search* (PBS). As the main strategy of A* is to keep opening the most promising node, we can consider all nodes at the boundary of the flood area "most" optimal (although this is not always strictly true). Whenever we hit a node flooded by an opposite core, we can immediately complete the path using the alternate core's pathfinding metadata. There is no need for expensive mutexes; both cores can just check a shared "break flag" in main memory to see if they should stop because the other core found a collision or gave up.

An example path found using PBS is shown in Figure 1. On the left, we see that a collision was detected somewhere halfway, when both cores have performed, in parallel, virtually the same amount of work. Connecting the two "half-paths" together yields a path that is only slightly more expensive than the most optimal path. Most discrepancies relative to A* are not erroneous when taking the flow of the full path into account but rather the result of a different bias because of the reversed search direction. The insignificant loss of optimality is because PBS stops the search just a bit too soon, when it detects a collision.

DOI: 10.1002/cav

Figure 1. (Left) With the use of Parallel Bidirectional Search (PBS), two cores flood towards each other until a collision is detected. The full path is then constructed by linking both "halves" together at the collision node. (Right) PBS path overlaid with the optimal A* path, whereby the red cells denote path deviations. Clearly, much less nodes are flooded by PBS than by A*.

Often, the area around the collision node does not get fully flooded, so potentially there might be cheaper nodes in this area, which will no longer be discovered. However, this deviation is generally very low given a fairly uniform travel cost between neighboring nodes.

We can, therefore, conclude that, strictly speaking, PBS is no longer optimal, but it is still complete; that is, it will find a path if it exists. In worst-case scenarios where no collision occurs, PBS is basically reduced to a normal A* search.

Once you accept loosing strict optimality, we found that there is room for further performance improvements. As discussed in the previous section, with each pass through its *Now* list, FS utilizes a threshold value to determine which nodes to process. Normally, the lowest f(n) value is selected to imitate the same processing order of candidate nodes as a classic A* would. In this research, however, we found that by increasing the minimal threshold value by a fixed *ThresholdRelaxation* constant, we obtain significant performance improvements with only minor additional degradations in path quality. This threshold relaxation enables FS to open more nodes during each pass, which means we not only have more work carried out in parallel but also enable the algorithm to flood outwards faster, which in turn results in earlier collisions. By "artificially" incrementing the threshold, we basically decide to sort the *Now* list "less thoroughly," which may of course result in less optimal node selections during each pass. Throughout all our experiments, we empirically found out that a *ThresholdRelaxation* value of 5 proved to be the "sweet spot," resulting in a speedup factor of 1.5 up to 2.5 and still resulting in paths that were just 1% more expensive.

Figure 2 presents the control diagram of our implementation of the PBS algorithm.

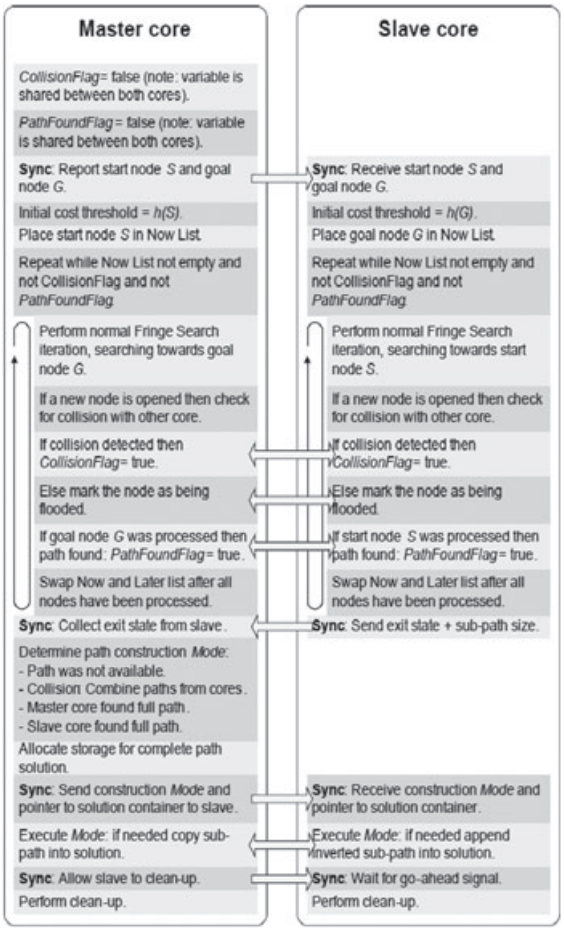

Figure 2. Detailed diagram of the Parallel Bidirectional Search algorithm.

2.2. Distributed Fringe Search

Our second attempt towards parallelization speedup was to use the FS's *Now* and *Later* lists to literally distribute "work" among multiple CPU cores. As mentioned previously, during each pass through the *Now* list, FS performs some very simple tests to determine if new nodes should be processed. This processing mostly involves adding new nodes to the *Later* list. The main idea behind the new algorithm we came to call *Distributed Fringe Search* (DFS) is thus to distribute the *Now* list over all available cores, have them process their share, merge the individual *Later* lists, swap this with *Now* and start all over again. A nice feature of FS is that the *Now* and *Later* lists do not need any sorting; thus, distributing them is very easy. Also, each core can compute the smallest f(n) value it has found locally, so that the master core can collect them and only needs to do a few comparisons to determine the cost threshold that should be used next for all cores.

The main advantage of DFS is that it can effectively utilize more cores, being no longer limited to two cores, as PBS. Although this is a great strength, it is also its Achilles' heel. By distributing nodes "arbitrarily" over multiple cores, we have lost the ability to perform the "correction step" as a normal A* and/or FS implementation would do.

This is illustrated in Figure 3, where we see a very simple graph that is being flooded with A* in the direction from node A to node D, the actual goal being a node G somewhere far off. Beginning at step 1, we see that node A has been opened at some point in time. In step 2, A* decides to close it and open its neighboring nodes B and C. When these nodes are placed on the *Open* list, we connect them with their parent node A because currently these both suggest to be the shortest path available. Then, in step 3, it turns out that node B has the lowest f(n) value (we are being pulled in the direction of goal node G) and it is closed. By doing this, node D will be opened and linked back to its parent node B again. However, it turns out that traveling from B to D is actually very costly, but because A* relies on the $h(n)$ function to do its look-ahead guessing, this will not be noticed. With d(n) being the true cost between nodes n and G, we only need to guarantee that $h(n) \leqslant d(n)$ for A* to be admissible. If we take h(n) to be, for example, the Euclidean distance, then we might quite often strongly underestimate the true travel cost (especially when all sorts of artificial penalties have been applied). So when, in step 4, node C is finally closed, we actually need to check if there are any nodes in the *Open* and *Closed* lists that we should link to if that would result in a lower total travel cost; in this case, we need to appoint node C as the parent of node D.

Note also that after this "correction step," A* needs to sort its *Open* list again to make sure we always keep selecting the most promising node. For DFS, however, we cannot perform this correction because when closing a particular node, we never know for sure that neighboring nodes are also present in the part of the *Now* list that is being processed by the CPU core. Forcefully implementing the required A* correction step would require us to stall other cores while we access their flooding data, which is of course very detrimental and should thus not be attempted. As a result of all these, it is no longer guaranteed that DFS will find the cheapest path; that is, DFS is not optimal, as we can clearly see in Figure 4.

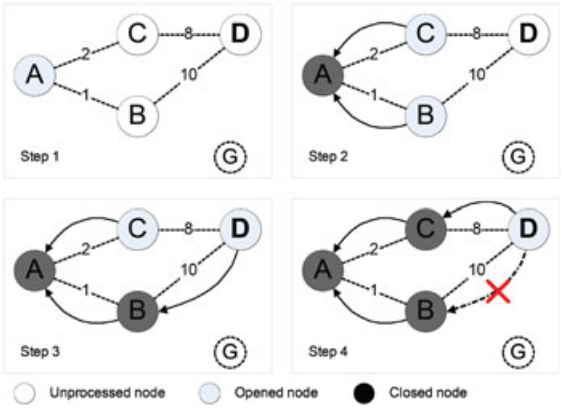

Figure 3. An example of how A* floods through a graph in the direction of goal node *G*. Each time a node is closed, we need to determine if meanwhile better paths have become available. This "correction step" is crucial to properly reconstruct the path later.

Another drawback of DFS is that it is also no longer possible to keep track of the "parent–child" relation of nodes during flooding (as illustrated previously for normal A*), although this information is needed to reconstruct the resulting path. The solution, therefore, is to use (i) a shared buffer in which all cores write to signal flooded nodes to each other (for which we only need to raise Boolean flags) and then (ii) a separate "private" buffer for each core to store g(n) values to evade race conditions that might otherwise actually prove problematic. The final path can then be correctly reconstructed by starting at the goal node G and then searching our way back to the start node S by repeatedly traversing towards the neighboring node with the lowest g(n) value found in any of the private buffers. Although this sounds discouraging, it is, in practice, not critical; we found surprisingly few cases of "over-flooding" (nodes tagged "Multiple Cores," in Figure 4(left)). Mostly, nodes only have a single g(n) computed for them, so no expensive floating-point comparisons are needed to find the lowest one.

The overall load balance seems fairly good (see Figure 4(left)), although we have noticed that there is always one core that seems to be doing most of the work. That core has often flooded most of the areas in which the final path was found, suggesting that this is likely due to the A* heuristic function: this function is designed to pull the search towards the goal node, and as long as this goes on "unhindered," it will always favor nodes for that particular core. The other cores will often be searching through "branches" elsewhere that later on turn out to be dead ends.

DOI: 10.1002/cav

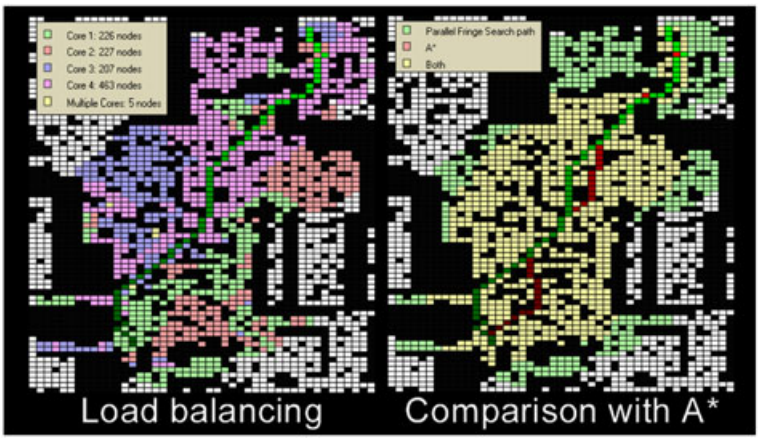

Figure 4. (Left) The load balance achieved by Distributed Fringe Search. (Right) With the loss of the corrective property, Distributed Fringe Search ends up with less than optimal paths. Note that the deviations with the optimal A* path are also just a matter of different bias; the actual additional path cost was only roughly 2%.

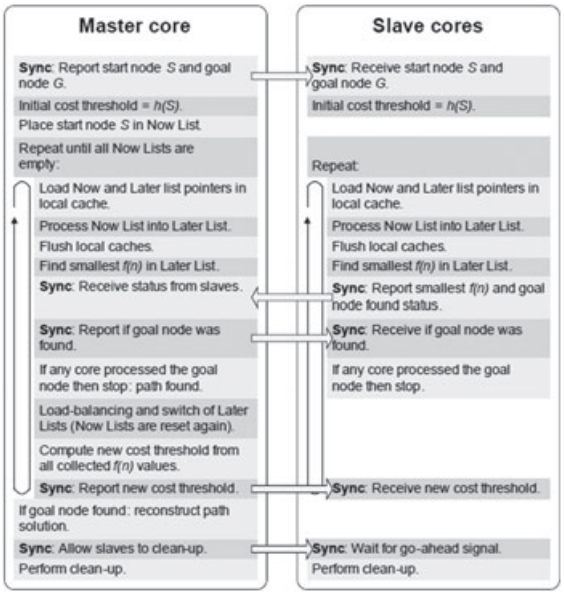

Figure 5. Detailed diagram of the Distributed Fringe Search algorithm.

Figure 5 presents the control diagram of our implementation of the DFS algorithm. Conceptually, the DFS approach sounds promising because it allows us to distribute the workload quite naturally over all available cores. In practice, however, the results obtained are less spectacular. After some profiling, it turned out that a significant amount of time is still wasted on cores waiting for each other, suggesting that the load balancing is still far from optimal.

2.3. Parallel Hierarchic Search

Another attractive way to utilize multi-core architectures is to have each core find small segments of the total path. Small searches whereby the segment's goal node is relatively close to its start node are significantly faster because far less nodes will become flooded. To do this, however, we need to guess where some *way-points* will be located in the search space so that, as it were, we can "connect the dots" between them. This can only be properly carried out if we employ a high-level graph representation of the actual graph to search through. With this high-level graph, we can roughly guess how the full path will traverse the search space and obtain way-points from it, hence the name *Parallel Hierarchic Search* (PHS).

There are many techniques to obtain such high-level graphs, ranging from manually adding way-points to automated schemes such as "Probabilistic Roadmap Method" [3,10,11]. For our PHS implementation, we created a grid randomization algorithm with a top-down approach, which generates "chambers" that are linked with smaller corridors, which are then filled with randomly placed obstacles; see Figure 6 for an example. This enabled us to easily generate many correct high-level hierarchies so that we could run large test batches.

Each node in the high-level graph is linked to a corresponding anchor node in the actual graph that needs to be searched. The first step in the PHS algorithm is to find a path through the high-level graph and then finding subpaths connecting the consecutive anchor nodes. Doing this will, however, never results in an immediately natural looking path because the anchor nodes might be needlessly off course. So a "beautification" step is applied by constructing new way-points halfway at the found path segments (by just picking the middle node of the path-segments sequence of solution nodes). The idea behind this is that it will help us find "shortcuts" between the high-level waypoint anchor nodes. We found that just a single beautification iteration already yields quite acceptable results.

Parallelizing the algorithm is basically a matter of having the master core generate the way-points and then letting

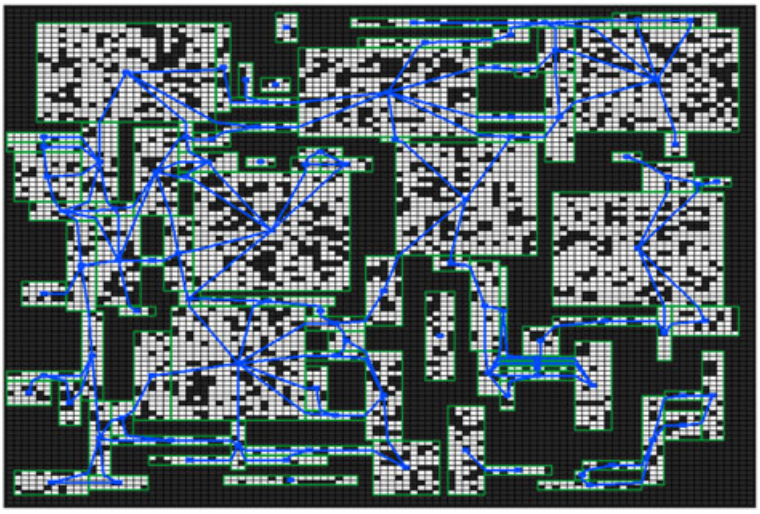

Figure 6. An example of our randomly generated search space, consisting of interconnected "chambers." The high-level graph is represented in blue.

all the cores try to construct the path segments. To enable the cores to gain access to the path segments information buffer, we are forced to employ a more expensive "critical section" that is provided by the operating system. This will allow safe access to selecting a path segment and returning a pointer to found path segment solutions (cores are not allowed to clear these solutions until all processing has been completed, so that the master core can safely access them). We can give each core its own copy of the graph so that there will be no cache collisions during the searches themselves. Figure 7 presents the control diagram of our implementation of the PHS algorithm.

Figure 8 shows an example of a path generated using the PHS algorithm. We can clearly see that PHS only floods nodes in the near vicinity of the final path and does not fan out into a "leaf"-shaped flood space as A* would do. The algorithm was about 1.6 times faster than a classic A* approach, but this came with a penalty: the resulting path has a noticeably higher cost and has a less "smooth" appearance. In the middle of the figure, we can see that the high-level path is distanced quite far away from the optimal path, as could be expected. Finally, on the right, we see that the load balancing is fairly acceptable. Many nodes have been flooded by multiple cores, but this is again due to the second phase smoothing. Because cores will process new path segments when they are done with the previous one, we will see some nodes being flooded by different cores in different phases.

In general, the results obtained with PHS are rather disparate. In some cases, we can obtain very good speedups, and in other cases, we do not. The overall path quality leaves much be desired; often, the paths will stray quite a bit from the optimal path, further contributing to increase the search duration.

3. ALGORITHM EVALUATION AND COMPARISON

In all preceding experiments, no special attempts have been made to "optimize" the parallel algorithms, other than those dictated by common sense. Moreover, they all use established basic libraries, such as the C++ Standard Template Library, expressly chosen for their stability rather than for their performance.

We obtained our measurements from a total of 2000 samples, by finding 100 random paths in 20 random maps. For each sample, we measured the time taken to find a path between two randomly selected nodes from a 400 by 400 eight-way connected uniform grid, internally represented by a directed graph. A Euclidean distance was used as A* search heuristic. Each sample was repeated five times for each algorithm so that cache content would "stabilize." The best result of all the taken samples was then taken as the ultimate measurement result. All threads and processes were running on highest priorities. All samples have been taken on a 2.4 GHz Intel Core2 Quad CPU running Windows XP Pro SP2.

Because FS plays such a prevalent role in our experiments, wherever possible we have used this algorithm as an A* alternative. Therefore, FS has also been included in all measurements so that we can clearly tell if the speedup is due to the parallelization and not just the fact that FS was used instead of a classic A* implementation.

The measurement results are shown in Figure 9. For very short paths, the results are mixed, which means that the parallelization overhead is probably too high compared with the amount of work that has to be carried out. As the path length increases, the parallelized algorithms start to outperform the classic A* implementation extensively. Still, PHS

DOI: 10.1002/cav

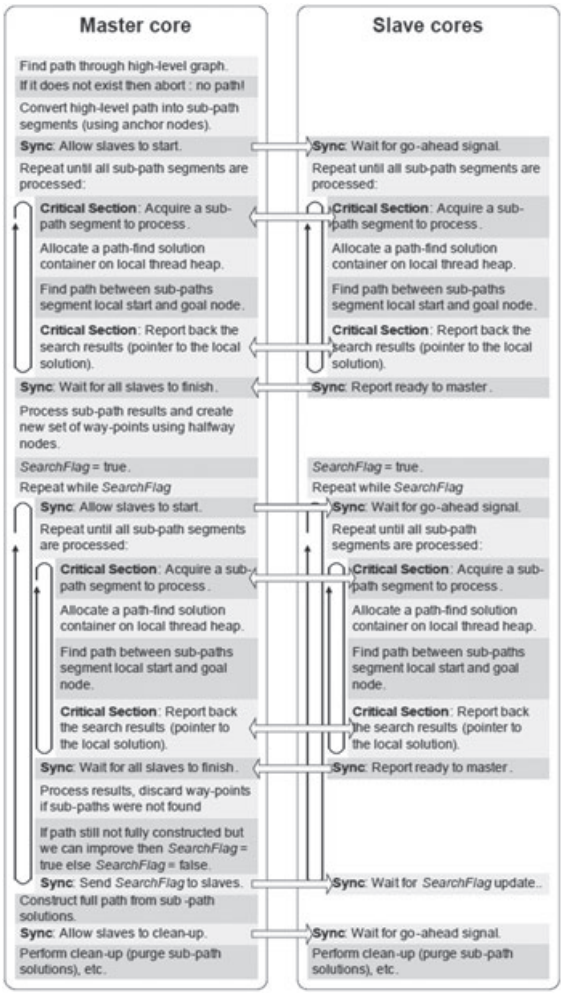

Figure 7. Detailed diagram of the Parallel Hierarchic Search algorithm.

is the "main loser," as it has the worst overall performance, probably because of requiring a mutex and "beautification" iteration. It has also by far the worst path cost overhead, ranging from 20% up to 45%.

For DFS, we can conclude that it is only worthwhile on longer paths; otherwise, the normal FS implementation still (slightly) outperforms it. Apparently, it is only for the longer paths that cores manage to get work done without interfering too much with each other's caches, which makes sense, because the flooded areas will be much larger and further spaced apart. DFS can yield up to 2,2 speedup relative to a classic A* implementation. As expected, its qualitative output is hampered by the fact that DFS has lost its corrective property in a far more significant degree than that of PBS: it has up to 4% additional path cost.

For PBS, the loss of its corrective property is only accumulated near the area of the collision node, which is generally very small. This clearly makes PBS a winner on all fronts: up to an impressive speedup of 6,7 relative to A*, while generating paths that, on average, are less than 1% more expensive than the optimal A* path.

The analysis of all these results is summarized in Table I. We conclude that

(1) Cache penalties have by far the largest impact on these algorithms' performance on multi-core architectures. The longer we can prevent cores to flood nodes in each other's areas, the better the performance will be. PBS clearly does this best because both cores start their search at the maximum possible distance apart from each other.
(2) High-level graphs tend to "malform" paths interpolated from them and require quite some "post-processing" to smoothen them out.
(3) High-level paths can seriously thwart the path finder when dynamic obstacles are in the way.

4. PARALLEL RIPPLE SEARCH

In this section, we present a novel algorithm called PRS, which capitalizes on the results of the experiments mentioned previously, to combine the strengths of all those algorithms while minimizing their weaknesses.

Algorithm

Parallel Ripple Search requires a high-level graph to guess where the final path will be located in the search space and uses it to position the cores at roughly equidistant way-points. However, in contrast to PHS, the cores will now find path segments by doing a normal A*-like flood towards their nearest neighbors instead. Like ripples in a pond, at some point their flood boundaries will overlap, and we can use these collisions to link the path segments together into the full path; see Figure 10.

The high potential of PRS is that when we find enough collisions, we can "short-circuit" and find connecting paths through previously flooded areas (for which we know that a path must exist). There is a good chance that these areas might still be (partially) lingering in a core's cache, so that accesses can be fast. This algorithm is also able to deal with dynamic obstacles much better than PHS could. If way-points turn out to be (partially) blocked, then this will just mean that adjacent ripples will not collide (not now, at least). So although it might take longer before a collision occurs with a ripple located further away, we will no longer run the risk of pulling the path in weird directions. Another advantage of this new approach is that we can utilize many more cores, basically one for each segment of the high-level path, thus overcoming the main restriction of PBS.

The cores that flood from the start node S and goal node G process what we could call the "essential" ripples: we

Figure 8. A path generated by the Parallel Hierarchic Search algorithm. (Left) Substantially less nodes have been flooded than with classic A*, but the resulting path is not as optimal and smooth. (Middle) The high-level graph and path used to form the resulting path. (Right) The flooding progress per core.

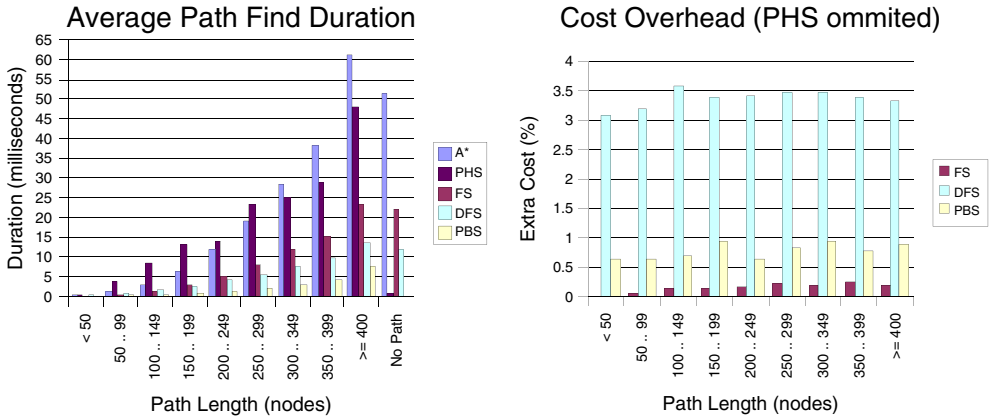

Figure 9. (Left) Overview of average pathfinding duration. Note how Parallel Bidirectional Search (PBS) can outperform a classic A* implementation almost seven times. (Right) Overview of average path cost overhead, relative to the length of the optimal path found by classic A*. (Parallel Hierarchic Search (PHS) results are omitted, as they can raise up to 45%). FS, Fringe Search; DFS, Distributed Fringe Search.

call them the *essential* cores, as opposed to all other *non-essential* cores. In a worst-case scenario, whereby none of the non-essential ripples ever collide with the two essential ones, PRS will basically have degraded to a PBS, which was shown to perform very well.

The algorithm is roughly described by the following steps:

- Find a high-level path P between start node S and goal node G.
- Two cores are assigned to the way-points at both ends (S and G) of path P.
- Depending on the number of edges in path P, we try to assign other cores at fairly equidistant way-points; these cores will form the "non-essential ripples."
- Phase 1: All cores start flooding the search space until enough collisions have been found to form a complete path:
  - Essential cores search towards each other's local start node (basically like PBS).
  - The remaining non-essential cores search towards the local start nodes of their direct neighbors.
- The master core will examine all the reports from the cores and determines which cores need to generate their path segments between which collision nodes. Note that some cores might have become superfluous or may need to be linked in a non-sequential order (it can happen!).
- Phase 2: All relevant cores construct their local paths and report these back to the master core.
- The master core assembles the final path.
- All cores perform a final cleanup.

DOI: 10.1002/cav

Table I. Summary of strengths and weaknesses of the three parallel pathfinding implementations.

	PBS: Parallel Bidirectional Search	DFS: Distributed Fringe Search	PHS: Parallel Hierarchic Search
Speed increase over classic A*	2,5–6,7	1,6–2,2	1,2–1,6
Path cost overhead	< 1%	3%–4%	20%–45%
Scalability	Bad: two cores only	Two or more	Two or more (but potentially much more effective than DFS)
Extra memory required	Low	High	Medium
Load balancing	Very easy: both cores always run at full speed	Hard, so we are always bound by the slowest CPU core	Medium, the quality of the high-level graph will automatically improve this
Cache "friendliness"	Very friendly	Very unfriendly	Fairly friendly
Implementation	Very easy and intuitive	More involved, needs many more synchronization moments, requires special techniques to optimize	Fairly easy (closer to "classic" parallelization).
Other	Any A* variant can be used	–	Requires high-level graph

During phase 1, the essential cores basically perform a normal FS towards the way-points of their neighbor cores. Their searches start, of course, at either start node S or goal node G, and not at the start or goal way-points of the high-level path. The search will continue until either a path can be constructed or there are no more nodes available, which means we have to give up. Collisions with the "non-essential" cores are analyzed to determine if a full path can be constructed, but they will not stop the cores. With some luck, we might be able to bypass some non-essential cores or maybe link up directly with the other essential core itself.

The "non-essential" cores use a slightly different heuristic for cost estimation function. This function is initially biased to flood towards the local start nodes of the adjacent cores:

$$\mathrm{h}(n) = \min(\mathrm{Estimate}(LocalS_{i-1}, n), \mathrm{Estimate}(LocalS_{i+1}, n))$$

Estimate() is a cost estimation function, such as the Euclidean distance. As soon as a first collision with a direct neighbor has been detected, we switch to a normal heuristic that will only flood in the opposite direction, towards the other neighbor's local start node. Once we have collided with that one as well, but determined that a full path is not yet constructed, we just keep flooding with the original heuristic function again. This will make the flood boundary expand in all directions again that in turn might find other collisions that prove to be more beneficial. Only when a non-essential core runs out of nodes will it abort the search. This event does not explicitly have to be reported back to the master core in any way; it might just mean that our initial guess using the high-level path was "wrong" and that the non-essential core started its search in an area that became isolated because of dynamic obstacles.

Once enough collisions have been detected and the master core has determined which cores will take part in the full path, we can start phase 2. During this phase, it is up to the corresponding cores to construct their local sections of the final full path. Synchronization between cores can all be carried out using spin locks to ensure that there is no unintentional operating system overhead. Now, for essential cores, it is very easy to construct their local path segments. As discussed earlier, the A* algorithm keeps track of a "parent node" for each node that it floods to link back towards its original start node. We thus only need

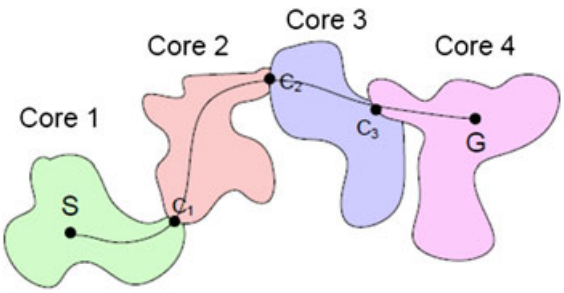

Figure 10. Symbolic illustration of *Parallel Ripple Search*: a path is constructed by first flooding the search space between start node S and goal node G at "equidistant" positions. As soon as enough "ripple collisions" have occurred, we can use them to construct the full path.

to look up the collision node and follow the parent links back to what will either be the original start node S or goal node G.

For non-essential cores, we need to employ a different approach. We cannot use the parent links because these will always lead us back to the local start node at the location of the corresponding high-level path anchor node. As discussed previously for PHS, we need a smoothing phase in an attempt to "iron out" the potential outlier that is the high-level path node itself. Especially if dynamic obstacles blocked our way, we need to make sure that the high-level path will not pull the resulting path way off course. For this, we do a new pathfinding session to find a suitable path between the two collision nodes that will ultimately "bridge the gap" between the neighbors of the core. We can, however, significantly speed up this process by simply limiting the flood area to nodes that have been flooded before. This information is often directly available via "visited flags" in the nodes themselves, and thus, the only extra cost is the pathfinding session itself. Note that because the flood areas will ideally be relatively small, a large amount of data will already be in the core's cache, thus making such re-visits a very fast process. Once all local path segments have been obtained, the master core simply needs to copy them into a single buffer, making sure that no duplicate entries from the collision nodes are copied.

The overall implementation time of PRS was significantly longer compared with PBS, but not as long as that of DFS, provided that we already have the means for generating high-level paths. PRS requires more synchronization moments, and we do need to keep track on how to safely access memory without having race conditions causing real problems. The cores always share one memory pool in which they will write their unique core IDs for each node they flood. In this way, other cores can detect when they trod on each other's toes and handle collisions. If we limit the size of the IDs to a single byte, we can be sure enough that writing them is "atomic" and the chance on race conditions is actually very small. And even then, if this does happen, the cores are bound to detect the collision during their next iterations as they further flood into each other's "body mass." For more details, please refer to Figure 11, which presents the control diagram of our implementation of the PRS algorithm.

Figure 12 shows an example of a path obtained with the PRS algorithm, 60% faster than a classic A*. The differences between both paths are small, mainly because the high-level path has managed to make a very good "guess."

Parallel Ripple Search is also more robust and can much better handle unexpected obstacles on the high-level path, as shown in Figure 13. If dynamic obstacles block choke points such as doorways, then PRS will circumvent these automatically, because any non-essential core that is flooding the area behind it will take longer to collide with other cores while it is attempting to "find its way out." This delay in collisions provides a "natural optimization effect," in the sense that other cores that are positioned more favorably are more likely to collide first and, thus, bypass the

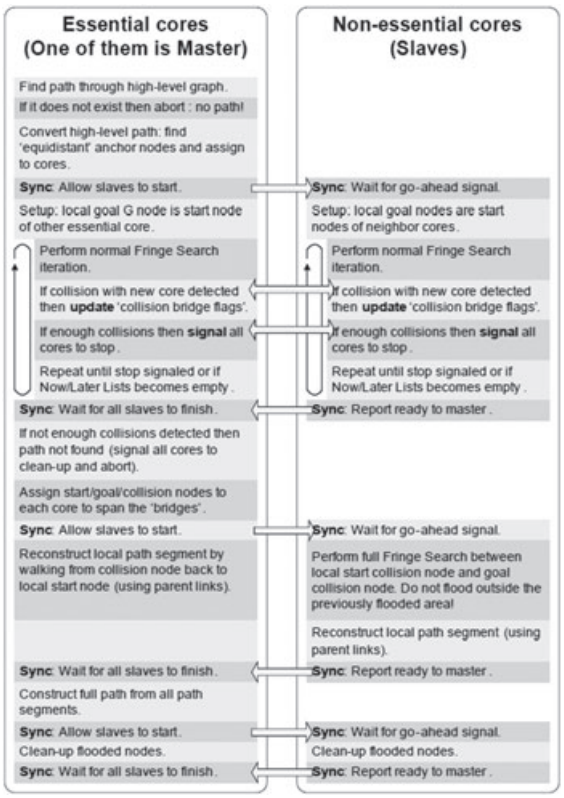

Figure 11. Detailed diagram of the Parallel Ripple Search algorithm.

unfavorably positioned ones. As pointed out earlier, in extreme cases, whereby a non-essential core is flooding a completely isolated area, no core will ever collide with it, and we can even abort its futile search attempts when enough collision have already been found to construct a full path. The local path findings during phase 2 also enable us to deal with dynamic obstacles efficiently and considerably smoothen out the resulting path. The secondary effect of this effect is that it will suppress a "gravitation" towards the anchor nodes of the high-level path (which proved so problematic for PHS). Granted, our approach gives no absolute guarantees that all artifacts are most optimally dealt with. However, throughout our experiments, we empirically ascertained that mostly the heuristic function very favorably directs the flood boundaries.

Finally, the quality of the paths found is in general good, although they are still influenced by the route and orientation of the high-level path.

5. PERFORMANCE

We have repeated for PRS the same experimental measurements previously described for the other parallel algorithms. As PBS was clearly the best alternative so far, here we will limit the discussion to FS, PBS and PRS to keep a clear overview.

Comp. Anim. Virtual Worlds 2012; **23**:73–85 © 2012 John Wiley & Sons, Ltd.
DOI: 10.1002/cav

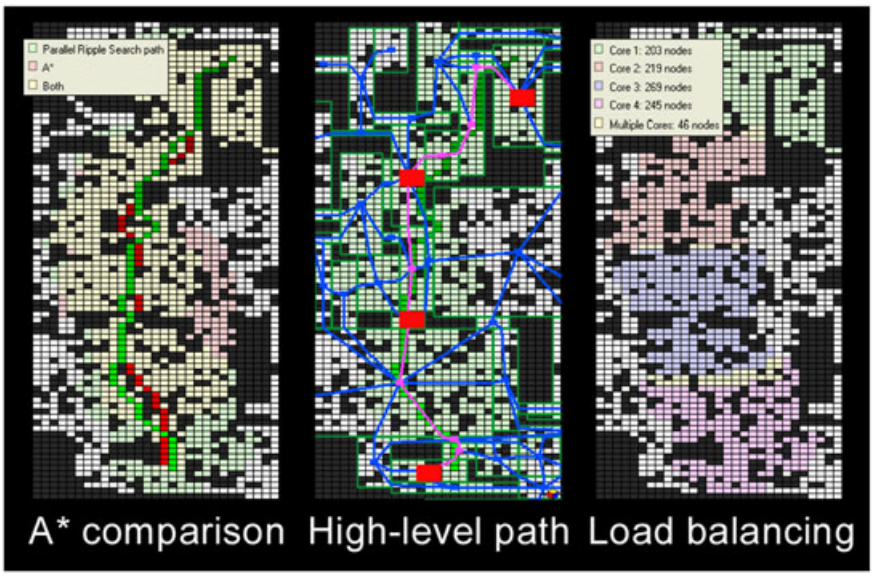

Figure 12. Path generated with the Parallel Ripple Search algorithm. (Left) Comparison with the classic A*. (Middle) The high-level path used and the way-points used to start off the non-essential cores. (Right) Each core has flooded roughly the same amount of nodes. The yellow nodes indicated collisions.

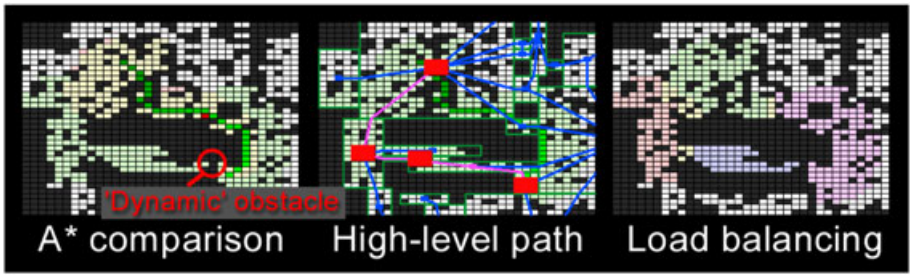

Figure 13. Example of how the Parallel Ripple Search algorithm still manages to bypass unexpected obstacles on the high-level path (middle). In this case, two cores did not manage to collide with each other (right), and thus, only the other cores contributed to the resulting path. A comparison with the classic A* implementation (left) shows that both paths were virtually identical.

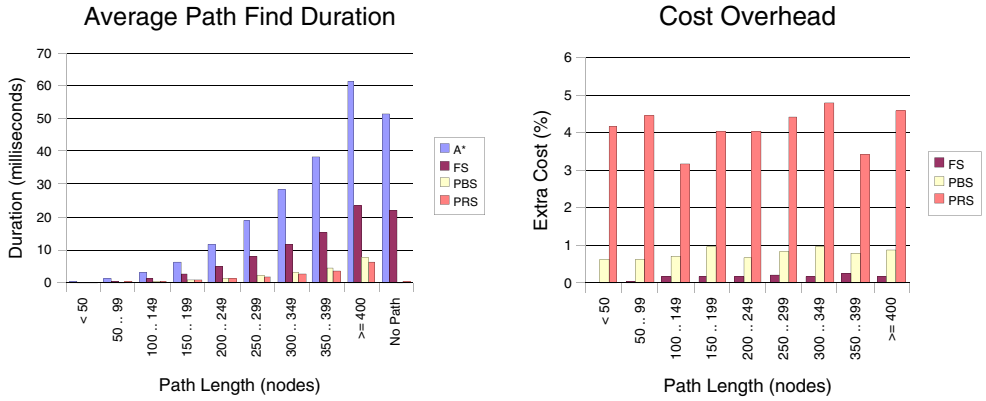

Figure 14. (Left) Comparison of average pathfinding duration. (Right) Comparison of average cost overhead. The cost overhead remains fairly constant for all variants.

From Figure 14, we conclude that PBS is still a very strong candidate on short to medium path lengths. Above roughly 200 nodes, PRS finally starts to capitalize on the fact that it can utilize more than two cores, at which the speedup factor is in the range 2,5–10 compared with classic A*. Up to that point, PRS has too much overhead and/or cannot utilize all its cores when not enough way-points are found in the high-level graph. Regarding path quality, we can see that PRS generates paths that are on average about 4% more expensive. This deviation is partially because the algorithm relies on the collisions to be favorable (which is, of course, not always the case), and also on the use of the FS *ThresholdRelaxation* constant, so that it expands faster outwards [12]. Also, in contrast to classic A*, we do not explicitly search for better nodes around collision nodes. The cost overhead for PBS is very low, less than 1% on average, which makes it an excellent alternative for short to medium length paths. The fact that PRS seems to have a "constant" overhead factor indicates that although the algorithm is "probabilistic" in nature, it is still able to make good enough "guesses" on a consistent basis. This is directly linked to the quality of the high-level graphs, which is therefore an essential component of any successful PRS implementation.

6. CONCLUSION

We have implemented a number of parallel pathfinding algorithms to investigate their behavior and performance in multi-core architectures. We concluded that all these algorithms exhibit one or more weaknesses; for example, they have a large overhead, yield far from optimal paths, do not easily scale up to many cores or are cache unfriendly. The latter was found to be crucial, as currently available multi-core CPUs have good cache look-ahead prediction, as long as shared pages do not get written into too often. In other words, for these architectures, data separation is key to efficient pathfinding.

In this article, we proposed PRS, a novel parallel pathfinding algorithm that largely solves the aforementioned limitations. Basically, the algorithm employs (i) two "essential cores" to flood at the path extremities (like PBS does) and (ii) all other available "non-essential cores" to flood local search areas, starting at "equidistant" intervals on a high-level path. These cores then use A* flooding behavior to expand towards each other, yielding good "guesstimate points" at border touch on. As a result, all cores effectively run at full speed until enough way-points have been found.

Like most other parallel algorithms, PRS sacrifices some path quality for speed: it runs roughly 2,5 up to 10 times faster than a classic A* implementation, with only an average minor penalty of 4% in path cost. This inevitable loss of optimality justifies the use of the FS variant, which is instrumental to further improve performance by means of its threshold relaxation: not only is more work carried out in parallel, it also expands flood boundaries faster, resulting in earlier collisions.

The PRS algorithm does not rely on any expensive parallel programming synchronization locks or mutexes but instead relies on the opportunistic use of node collisions among cooperating cores, exploiting the multi-core's shared memory architecture. As a result, PRS is easily portable to different platforms that provide symmetric multiprocessing architectures and/or embedded systems that do not provide concurrent programming primitives other than threads.

Future research should focus on, at least, two directions. First, it would be worthwhile improving the quality of the high-level path, enabling PRS to make better guesstimates on where the non-essential cores should best start flooding from. Second, new performance gains should be achieved by further reducing cache collisions between cores, for example, by re-arranging the memory location of nodes to better reflect their real-world topology.

In conclusion, the PRS algorithm (i) is a fast and practical pathfinding solution for large and complex maps, (ii) it flexibly handles dynamic obstacle in a natural way, and (iii) it guarantees good scalability facing the increasing amount of cores of present day hardware.

REFERENCES

1. Sniedovich M. Dijkstra's algorithm. The University of Melbourne, Australia. http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html.
2. Patel AJ. Variations of A*. *Amit's Game Programming Site*, 2004. http://www.dc.fi.udc.es/lidia/mariano/demos/amits/theory.stanford.edu/~amitp/gameprogramming/variations.html.
3. Björnsson Y, Enzenberger M, Holte R, Schaeffer J. Fringe Search: beating A* at pathfinding on game maps. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*, Essex, 2005; 125–132. http://www.cs.ualberta.ca/~games/pathfind/publications/cig2005.pdf.
4. Bleiweiss A. GPU accelerated pathfinding. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2008*, Sarajevo, 20–21 June 2008; 65–74.
5. Buluç A, Gilbert JR, Budak C. Solving path problems on the GPU. *Parallel Computing* 2009. DOI: 10.0106/j.parco.2009.12.002.
6. Cohen D, Dallas M. Implementation of parallel path finding in a shared memory architecture. Department of Computer Science Rensselaer Polytechnic Institute, Troy, NY, 2010. http://www.cs.rpi.edu/~dallam/Parallel.pdf.
7. Cvetanovic Z, Nofsinger C. Parallel Astar search on message-passing architectures. System Sciences.

In *Proceedings of the Hawaii International Conference on System Science,* Vol. 1, Hawaii, 2–5 January 1990; 82–90.
8. Sturtevant NR, Geisberger R. A comparison of high-level approaches for speeding up pathfinding. In *Proceedings of AIIDE 2010 - 6th Conference on Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA, USA, 11–13 October; 76–82.
9. Brand S. Efficient obstacle avoidance using autonomously generated navigation meshes. *MSc Thesis*, Delft University of Technology, The Netherlands, 2009.
10. Amato N. Randomized motion planning. Texas, USA, 2004. http://parasol.tamu.edu/~amato/Courses/padova04/lectures/L8.prms.pdf.
11. Nohra J, Champandard AJ. The secrets of parallel pathfinding on modern computer hardware, *Intel Software Network,* Intel Corp, 2010.
12. Botea A, Muller M, Schaeffer J. Near optimal hierarchical path-finding (HPA*). Department of Computing Science, University of Alberta, 2006.

AUTHORS' BIOGRAPHIES

Sandy Brand received his MSc in Computer Science from Delft University of Technology, The Netherlands, in 2009. The research presented in this article stems from a "slightly escalated side project" during his graduation work [9]. His current research interests include behavior trees and motion planning. Sandy has over 10 years of professional game development experience, having worked on international titles at several game studios. He is currently a senior artificial intelligence programmer at Crytek, Germany.

Rafael Bidarra graduated in Electronics Engineering at the University of Coimbra, Portugal, in 1987, and received his PhD in Computer Science from Delft University of Technology, The Netherlands, in 1999. Rafael is currently an associate professor of Game Technology at the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology, where he leads the game technology research lab at the Computer Graphics and Visualization Group. His current research interests include procedural and semantic modeling techniques for the specification and generation of both virtual worlds and game play, semantics of navigation, serious gaming, game adaptivity and interpretation mechanisms for in-game data. Rafael has published numerous papers in international journals, books and conference proceedings, integrates the editorial board of several journals and has served in many conference program committees.