2013 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS)
November 3-7, 2013. Tokyo, Japan

Dynamic Search on the GPU

Mubbasir Kapadia[1,4], Francisco Garcia[2], Cory D. Boatright[1,3], and Norman I. Badler[1]

[1]University of Pennsylvania
[2]University of Massachusetts Amherst
[3]Grove City College
[4]Disney Research, Zurich

Abstract— **Path finding is a fundamental, yet computationally expensive problem in robotics navigation. Often times, it is necessary to sacrifice optimality to find a feasible plan given a time constraint due to the search complexity. Dynamic environments may further invalidate current computed plans, requiring an efficient planning strategy that can repair existing solutions. This paper presents a massively parallelized wavefront-based approach to path planning, running on the GPU, that can efficiently repair plans to accommodate world changes and agent movement, without having to restart the wavefront propagation process. In addition, we introduce a termination condition which ensures the minimum number of GPU iterations while maintaining strict optimality constraints on search graphs with non-uniform costs.**

I. INTRODUCTION

Pathfinding is a fundamental problem in robot navigation, with a large variety of proposed approaches [1], [2] that balance computational performance, problem domain complexity, and plan optimality. Graph-based search methods such as A* [3] provide strict optimality guarantees but cannot handle dynamically changing environments. Real-time planners [4] have been proposed which provide anytime solution guarantees, and can efficiently repair existing plans to accommodate world changes and agent movement. However, these approaches are difficult to parallelize. Parallel search algorithms [5] exploit multiple computer resources to greatly reduce computational cost, but sacrifice optimality guarantees. Additionally, they have no mechanism to efficiently handle world changes and agent movement.

This paper presents a massively parallelizable, wavefront-based approach to path planning that can exploit graphics hardware to considerably reduce the computational time, while still maintaining strict optimality guarantees. This approach performs efficient updates to accommodate world changes and agent movement, while reusing previous computations. We introduce a termination condition which ensures that the plans returned are strictly optimal, even on search graphs with non-uniform costs, while requiring minimum GPU iterations. Furthermore, the computational complexity of our approach is independent of the number of agents (traveling to the same goal), facilitating optimal, dynamic path planning for a large number of agents in complex dynamic environments, opening the possibility to large-scale crowd applications. This paper makes the following contributions:

- A wavefront based search technique that can efficiently handle world changes and agent movement, while reusing previous efforts, and is amenable to massive parallelization.
- A termination condition which enforces strict optimality guarantees, even for non-uniform search graphs, while requiring the minimum number of GPU iterations.
- Extension to handle any number of moving agents traveling to the same goal, at no additional computational cost.

To our knowledge, this is the first massively parallelizable, dynamic search technique with strict optimality guarantees for non-uniform search graphs.

II. RELATED WORK

There exists considerable research in robot motion planning [1] that investigate a variety of trade-offs in computational cost, domain complexity, and solution optimality. Discrete search methods like A* [3] provide strict optimality guarantees but cannot efficiently handle dynamic world updates and agent movement. To meet real-time constraints, techniques are proposed which limit the horizon of the search [6], or reduce the effective branching factor by hierarchically coarsening the resolution of the problem domain [7]. AD* [4] is an anytime dynamic planner that satisfies strict time constraints while efficiently repairing existing solutions to accommodate dynamic events. Optimizations for lattice-based methods have been proposed [8] which prune transitions by embedding semantic information in the lattices. Tree-based search algorithms [9] have been optimized to exploit multiple processors. However, these approaches are not amenable to massive parallelization using graphics hardware.

GPU accelerated path planning algorithms provide tremendous performance boost, enabling the solutions of higher dimensional problem domains, but return sub-optimal paths. The work in [10] demonstrates shortest path calculations for graph-based searches on the GPU. The work in [11] uses a blocked recursive elimination strategy to utilize the computational power and memory bandwidth of the GPU. Randomized searches [12], [5] have been successfully ported to the GPU, by doing multiple short-range searches in parallel, but provide no optimality guarantees. Distance fields can be used to solve

multi-agent planning on the GPU [13]. Crowd simulation techniques [14], [15] exploit GPU hardware to accelerate local collision avoidance for crowds but do not handle global planning. Hierarchical planning approaches [16] perform map abstraction to adaptively subdivide the search space into smaller grids, each of which can be resolved in parallel. Wavefront based algorithms [17] are amenable to parallelization and have been demonstrated in a wide variety of problem domains [18], yielding substantial performance benefits over serial algorithms.

Comparison to Prior Work. Our work provides the benefits of both dynamic search techniques [4] and wavefront-based algorithms [17] to provide a search technique which is massively parallelizable and can efficiently update search efforts to accommodate dynamic world changes and agent movement. Many wavefront-based search techniques exist; however, these methods cannot efficiently handle dynamic environments and require the propagation to start from scratch when the environment changes. Our approach differs from traditional search methods, as we do not maintain a sorted list of open or closed states, which cannot be easily parallelized. In contrast, all states are self-contained and can be evaluated independently without any need for synchronization. Agent movement and obstacle movement is handled in a similar fashion to AD* by propagating state inconsistency, while exploiting GPU parallelization.

III. METHOD OVERVIEW

Our method relies on appropriate data transfer between the CPU and GPU at specific times. In the initial setup, the CPU calls $generateMap(rows, columns)$ which allocates $rows \times columns$ states in the GPU to represent the entire world. Initially, all free states s have an associated cost of -1, $g(s) = -1$, which represents a state that needs to be updated, while obstacles have infinite cost, $g(s) = \infty$. Given an environment configuration with start and goal state(s), ***computePlan*** is executed which repeatedly invokes ***plannerKernel*** (a GPU operation) until a solution is achieved. We keep two copies of the world map: one for reading state costs and the other for writing updated state costs. After each iteration (*i.e.*, kernel execution), the two maps are swapped. This strategy addresses the synchronization issues inherent in GPU programming, by ensuring that the main kernel does not write to the same map used for reads.

Once the planner is done executing, each agent can just follow the least cost path from the goal to its own state to find the generated plan. If an obstacle moves from state s to state s', we update the GPU map by setting $g(s') = \infty$ and $g(s) = -1$. This means that s' is now an obstacle and the cost for s is invalid and needs to be updated. In addition, we check the neighbors of s' and mark them as inconsistent if they had s' as their least cost predecessor. The planner kernel monitors states that are marked as inconsistent and efficiently computes their updated costs (while also propagating inconsistency) without the need for resetting the entire map. Agent movement (change in start) is also efficiently handled by performing the search in a backward fashion from the goal to the start, and marking the previous state as inconsistent to ensure a plan update. Algorithm 1 provides the pseudocode for ***computePlan***.

Algorithm 1 ***computePlan***$(*m_{cpu})$

```
m_r ← m_cpu
m_w ← m_cpu
repeat
    flag ← 0
    plannerKernel(m_r, m_w, flag)
    swap (m_r,m_w)
until (flag = 0)
m_cpu ← m_r
```

IV. GPU-BASED WAVEFRONT ALGORITHM

Existing graph-based search [3] guarantee optimality and work well for dynamic environments [4], however they are not amenable to massive parallelization. The wavefront algorithm [17] takes its name as an analogy of the way it behaves. It sets up a map with a initial state which contains an initial cost. At each iteration, every state at the frontier is expanded computing its cost relative to its predecessor's cost. This process repeats until the cost for every state is computed, thus creating the effect of a wave spreading across the map. Wavefront-based approaches are inherently parallelizable, but existing techniques require the entire map to be recomputed to handle dynamic world changes and agent movement. Figure 1 visualizes the wavefront propagation process in a simple environment.

Our Approach. Algorithm 2 describes the shortest path wavefront algorithm ported to the GPU. The planner first initializes the cost of every traversable state to a default value, $g(s) = -1$, indicating it needs to be updated. States occupied by obstacles take a value of infinity, $g(s) = \infty$, and the goal state is initialized with a value of 0, $g(s) = 0$. The planner finds the value g of reaching any state s from the $goal$ by launching a kernel at each iteration that computes $g(s)$ as follows:

$$g(s) = min_{s' \in succ(s) \wedge g(s') \geq 0}(c(s, s') + g(s'))$$

where $0 \leq c(s, s') \leq \infty$ is the cost of traversing from state s to s', and is used to encode regions of the environment which should be avoided (*e.g.*, rough terrain, dangerous areas), in addition to obstacles that cannot be traversed. This process continues until all states have been updated at which point the planner terminates execution. To address the concurrency problem inherent in a massively parallel application, we use use two maps, one as read-only m_r and the other write-only m_w. Each thread in the kernel reads the necessary values to calculate the cost of its corresponding state from m_r, and writes it to its given state in m_w. This ensures that the map we are reading from will not change as we are executing the kernel. Once the kernel finishes execution, we swap m_r and m_w, thus allowing the threads to read the most recent map while preventing race conditions.

(a) (b) (c) (d)

Fig. 1. Wavefront expansion process. (a) 3 iterations. (b) 11 iterations. (c) 15 iterations. (d) 18 iterations.

Algorithm 2 ***plannerKernel***$(*m_r, *m_w, *flag)$

```
s ← threadState
if s ≠ obstacle ∧ s ≠ goal then
    for all s' in neighbor(s) do
        if s' ≠ obstacle then
            newg ← g(s') + c(s, s')
            if (newg < g(s) ∨ g(s) = −1) ∧ g(s') > −1 then
                pred(s) ← s'
                g(s) ← newg
                { evaluate_termination_condition }
```

The kernel also takes as a parameter a `flag` which is set depending on the termination condition used:

Exit when goal reached. The flag is originally set to 1 before each kernel run. If we find that goal state was updated, that means we have a path to it and can terminate execution. We do so by changing the `flag` as 0. This will produce considerably fewer iterations but will not guarantee optimality on search graphs with non-uniform costs. Since each thread corresponds to only one state, only one thread is able to modify this flag and no race condition is possible.

$$\textbf{if}(s == goal)\texttt{flag} = 0$$

Exit when whole map converges. An alternate exit condition is to continue propagating until the whole map has been successfully updated with accurate g values. This will guarantee optimality but with a considerable increase in the number of iterations. The `flag` is set to 0 before each kernel run. If there is any update in a given iteration, `flag` is set to 1 thus ensuring that the planner keeps running until no further update is possible. In other words, the kernel will terminate only when the cost computation for the entire environment has converged. This will compute costs for unnecessary parts of the environment but will guarantee optimal solutions.

$$\texttt{flag} = 1$$

Minimal Map Convergence with Optimality Guarantees. The naive approach discussed previously does much more work than it is necessary to find an optimal path. For large environments, this is prohibitively expensive. We introduce a termination condition that can greatly reduce the number of iterations required to find an optimal plan in large environments with non-uniform search graphs. If at any iteration, we find that the minimum g-value expanded corresponds to that of the agent, this means that a path to that agent is available and any other possible path would yield a higher cost. To make sure that the agent state is expanded at each iteration (to compare to the other states expanded), we give it a g-value of -1 before each kernel run, marking it as a state that needs to be updated. To implement this strategy, it is enough to just adjust the condition that would set the `flag` that terminates the execution:

$$\textbf{if}(g(s) < g(start) \vee g(agent) = -1)\texttt{flag} = 1$$

Once the planner has finished executing, an agent can simply generate its plan by following the reverse of the least-cost path from the goal to its position. Figure 2 compares the different termination conditions on a simple environment.

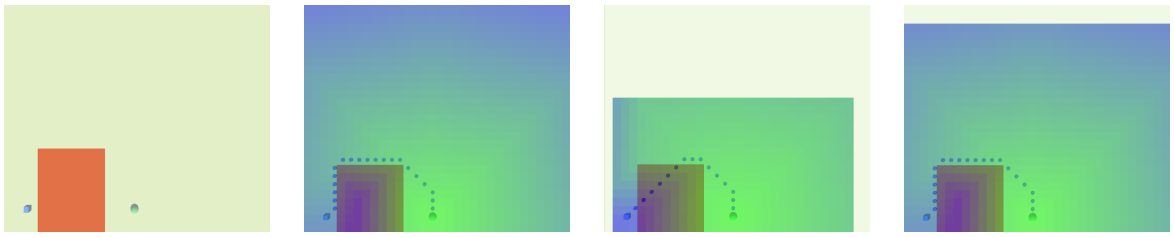
V. EFFICIENT PLAN REPAIR FOR DYNAMIC ENVIRONMENTS AND MOVING AGENTS

We extend the algorithm to handle dynamic environments where obstacle changes may invalidate plans that are currently being executed, and create a new plan if an agent diverges from a previous plan. To handle obstacle movements, we identify and resolve inconsistent states. A state is inconsistent if its predecessor is not the neighbor with lowest cost or if its successor is inconsistent. If an obstacle moves from state s to state s', then we set $g(s') = \infty$ and $g(s) = -1$ (marking it for update). We then run a kernel (Algorithm 3) which sets the g-value of every inconsistent state to -1. A state s with predecessor s' is inconsistent if $g(s) \neq g(s') + c(s, s')$. This kernel is executed repeatedly until all inconsistencies are resolved and is detected when there are no updates performed by any thread during an iteration. Algorithm 3 is triggered when environment changes are detected to ensure that node inconsistency is propagated and resolved in the entire map. Keep in mind that the following code will run in parallel, and that all read and write operations are done in two distinct maps. Keep in mind that the following code will run in parallel, and that all read and write operations are done in two distinct maps.

Algorithm 3 Algorithm to propagate state inconsistency

```
s ← threadState
if pred(s) ≠ NULL then
    if (g(s) == obstacle ∨ pred(s) == obstacle ∨ g(s) ≠ g(pred(s)) +
    c(s, s')) then
        pred(s) = NULL
        g(s) = −1
        incons = true
```

Handling agent movement is straightforward. For the non-optimized planner, the cost to reach every state has already



(a) (b) (c) (d)

Fig. 2. Comparison of termination conditions. (a) Non-uniform state space. The states shown in red are of much higher cost as compared to other states. (b) The planner terminates after updating g values for the whole map, producing an optimal path with significantly more iterations $= 17$. (c) Plan termination as soon as it finds a path to the goal, producing a sub-optimal path. Total number of iterations $= 8$. (d) Convergence of minimum number of states in the search graph, while ensuring path optimality. Number of iterations $= 12$.

been computed, so the agent would only require to reconstruct its path again. In the case of the optimized version of the planner it is necessary to run the planner again so that any state between the goal and the new agent position that has not been expanded, gets a chance to update its cost.

VI. MULTI-AGENT PLANNING

We extend our planner implementation by making a slight modification to the termination condition to account for multiple agents. We execute the kernel until all agent states have been reached, and the maximum g-value of a state that is occupied by an agent is less than the g-value of any other state that was updated during the current iteration.

$$\textbf{if}((g(s) < max_{a_i \in \{a\}} g(a_i)) \vee (g(a_i) = -1 \forall a_i \in \{a\}))$$

The number of iterations for convergence depends on the distance from the goal to the farthest agent. When the map is updated, each agent simply follows the least cost path from the goal to its position to find an optimal path. Note that our approach requires no additional computational cost to handle many agents, provided they share the same goal.

Multi-Agent Simulation. Since our approach can efficiently plan paths for a large number of agents, and use existing plans to efficiently repair solutions due to change in world state, we can easily extend our approach to simulate a crowd of autonomous agents. Each agent computes and maintains its path, and interleaves planning with execution by moving along its current path using a simple particle simulator with local collision-avoidance [19]. At each frame, the map is efficiently repaired to accommodate world changes and agent movement, thereby repairing the paths of all agents.

Multiple Target Locations. Our method can be easily applied to agents traveling between alternating goals, and would only require the agent to retrieve a path from the map corresponding to its current target location. However, we are limited to a small number of target locations, since a separate map needs to be maintained for each target, resulting in a significant memory overhead. Possible extensions to mitigate this issue include an adaptive quad-based environment representation, which would also reduce the computation of wave propagation in large worlds, and significantly reduce the memory footprint. Efficiently porting an adaptive-quad based environment representation to the GPU is the subject for future exploration.

VII. RESULTS

We ran our planner on several challenging navigation benchmarks [20] to showcase the benefits and limits over traditional methods (see Figure 3). We compared results using two different GPUs, specified in Table I. Our algorithm has an order of magnitude performance boost over CPU implementations of graph search and wavefront-based methods, which is even more significant for many agents and large environments.

TABLE I
GRAPHICS PROCESSING UNITS SPECIFICATIONS

Information	GPU 1	GPU 2
Type	Geforce GT 650M 2GB	GeForce GTX680
Warp Size	32	32
Threads/Block	1024	1024
Global Mem	2147483648 Bytes	2147483648 Bytes
MultiProcessors	2	8
Mem Clock Rate	900000 KHz	3004000 KHz
Mem Bus Width	128 bits	256 bits
Chip Clock Rate	950000 KHz	1058500 KHz

Figure 4(a) demonstrates the scalability of our approach with increase in number of agents on a 256×256 world map. We observe that there was no noticeable increase in the computational cost with increase in number of agents. Figure 4(b) illustrates the scalability of our approach to accommodate large environments. We tested it with a single agent with a goal distance of $N/2$ in a $N \times N$ world map. We observe that the use of the minimal yet sufficient exit condition (EXIT A) produces significant performance improvements over EXIT B as the planner does not have to wait until the g values of the whole map have converged, resulting in great savings.
Figure 4(c) illustrates the overall advantage of using our method with a simple test scenario where we handle obstacle, agent and goal movement. We generated a random map of size 512×512 populated with 8 agents. We can observe that our method took fewer iterations to reach an optimal solution

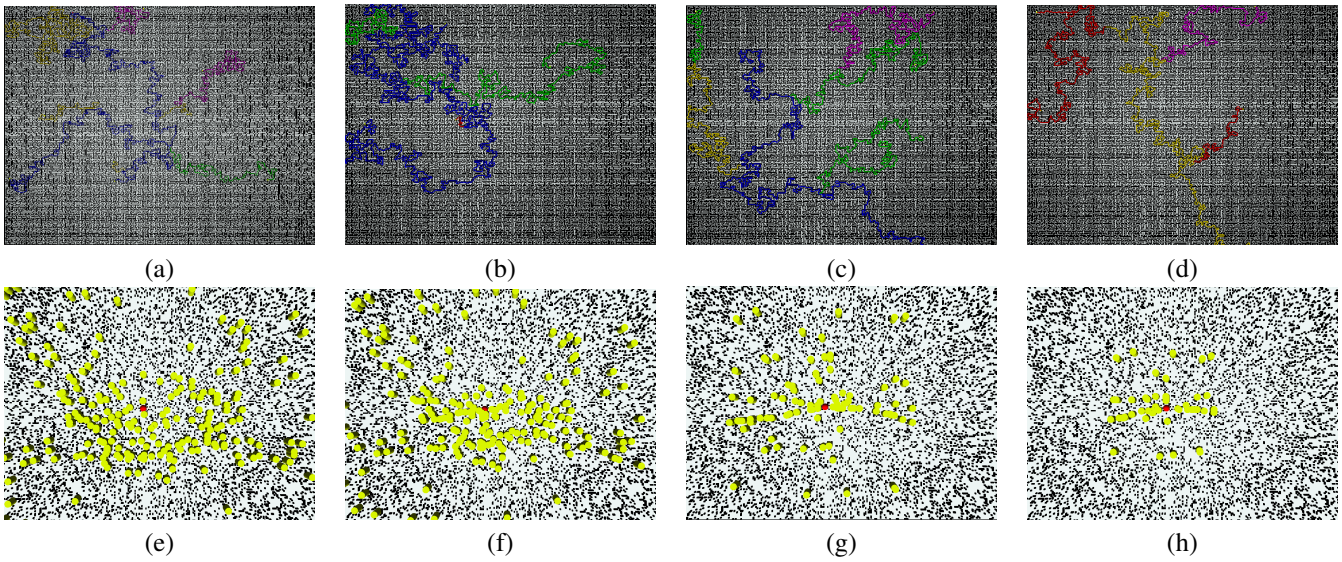


Fig. 3. (a)–(d) Global Navigation for multiple agents on a variety of challenging benchmarks [20] of size 512×512. Color lines are the computed paths, while a black region is an obstacle. (e)–(h) Global Path Planning and Simulation of 200 agents on a complex navigation benchmark.

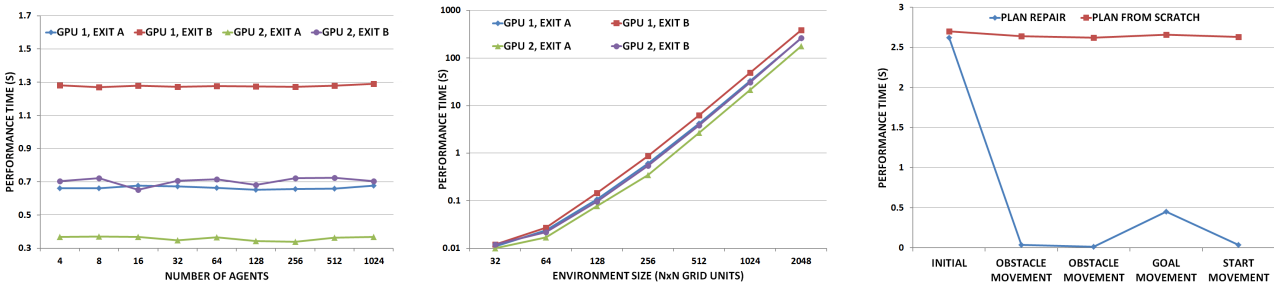


Fig. 4. (a) GPU planner performance with increase in number of agents. `EXIT A`: Exit condition that checks for convergence of only agent states. `EXIT B`: Exit condition which checks for convergence of whole map. All solutions returned are optimal paths. Experiment performed on a 256×256 environment. GPU memory = 5120 KB. CPU memory varies from 2048 KB to 2080 KB. (b) Time to compute optimal solution for different map sizes. (c) GPU planner performance for dynamic simulation with changes in environment, start, and goal.

TABLE II
ALGORITHM PERFORMANCE FOR DIFFERENT ENVIRONMENTS. (TIME IN SECONDS)

World Size	GPU 1		GPU 2	
	Exit A	Exit B	Exit A	Exit B
32×32	0.011	0.012	0.01	0.012
64×64	0.024	0.027	0.017	0.022
128×128	0.107	0.146	0.078	0.096
256×256	0.608	0.871	0.349	0.542
512×512	4.219	6.24	2.691	3.816
1024×1024	32.931	49.126	21.246	30.778
2048×2048	258.88	387.35	178.794	264.373

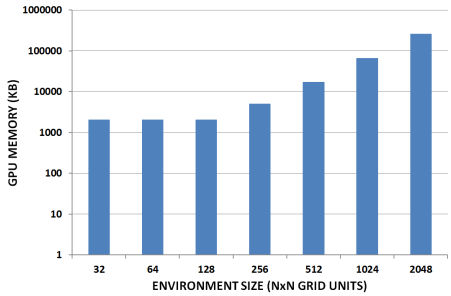


Fig. 5. GPU memory requirement with increase in environment size. GPU memory usage is 263 MB for a 2048×2048 environment.

at each step, with a significant performance improvement on the initial plan and after goal movement, when the map needs to be reset and planned from scratch. Figure 5 illustrates the memory requirements of our approach based on world size. Figure 3(e)–(h) demonstrates path planning for 200 agents in a randomly generated environment of size 512×512. Since our approach can efficiently handle dynamic updates, we can interleave planning with execution to create a crowd simulator.

Optimality. When the planner is first executed, state costs are updated until the start states of all agents are reached, and the maximum g-value of a state that is occupied by an agent is less than the g-value of any other state that was updated during the current iteration. During cost propagation, a state computes its g-value based by referencing its neighbor

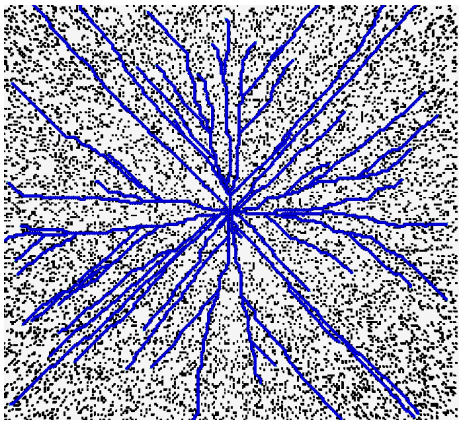


Fig. 6. Complex 512×512 with 200 agents. Goal is in center of map, and computed paths are shown in blue. Every black area is an obstacle.

with the least cost. A particular state is guaranteed to have converged when there is no other state with lower cost. This means that there cannot be a lower cost solution to reaching the state from the goal. Hence, this exit condition ensures that there will always be a valid least-cost path from the current position of all agents to the target. After an obstacle movement, if a state s or its predecessor s' contains an obstacle, or $g(s) \neq g(s') + c(s, s')$ then $g(s) = -1$, marking it for an update. This means that every state that defines an invalid transition will set its g-value to -1 allowing it to update in the next run of the planner. If there is a better plan for a given agent from the one we previously had, then newly updated states will have a lower g-value than the agent's start state. This means a lower-cost solution may possibly exist, and the planner continues executing until all inconsistencies are resolved or a least-cost solution is achieved.

VIII. CONCLUSION AND FUTURE WORK

We have developed a massively parallel wavefront-based planning technique which can efficiently handle world changes and agent movement by reusing previous computations. The computational cost of our approach is independent of number of agents, facilitating global path planning for hundreds and thousands of agents in very large, complex, dynamic environments. Furthermore, we demonstrate a prototype crowd simulator by interleaving planning with execution where the plans are efficiently updated to accommodate agent movement.

There are some limitations to our approach. A separate map needs to be maintained for each target location, resulting in substantial memory and computational overhead. This makes our current approach intractable for large numbers of agents with independent targets. One possible approach to attenuate the impact in memory would be to use a quad-based environment representation where open spaces can be represented as a coarser grid. Porting a quad-based environment representation to the GPU is ongoing work.

ACKNOWLEDGEMENTS

This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement # W911NF-10-2-0016. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. GPU hardware support from the NVidia Professor Partnership program is gratefully acknowledged.

REFERENCES

[1] J-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[2] M. Kapadia and N.I. Badler. Navigation and steering for autonomous virtual humans. *Wiley Interdisciplinary Reviews: Cognitive Science*, pages n/a–n/a, 2013.

[3] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, July.

[4] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proc. ICAPS*, 2005.

[5] J. Kider, M. Henderson, M. Likhachev, and A. Safonova. High-dimensional Planning on the GPU. In *Proc. IEEE ICRA*, 2010.

[6] Shawn Singh, Mubbasir Kapadia, Glenn Reinman, and Petros Faloutsos. Footstep navigation for dynamic crowds. *CAVW*, 22(2-3):151–158, 2011.

[7] M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano, and N.I. Badler. Multi-domain real-time planning in dynamic environments. In *Proc. ACM SIGGRAPH/EUROGRAPHICS SCA*, pages 115–124, 2013.

[8] J.J Kuffner. Efficient optimal search of euclidean-cost grids and lattices. In *Proc. IEEE IROS*, 2004.

[9] C. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proc. AAAI*, pages 128–132, 1988.

[10] D. Delling, A.V. Goldberg, A. Nowatzyk, and R.F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proc. IPDPS*, pages 921–931, 2011.

[11] A. Buluc, J.R. Gilbert, and C. Budak. Solving Path Problems on the GPU. 36:241–253, 2010.

[12] M. Gini. Parallel Search Algorithms for Robot Motion Planning . In *IEEE ICRA Current Approaches and Future Directions*, 1996.

[13] R.P. Torchelsen, L.F. Scheidegger, G.N. Oliveira, R. Bastos, and J.L.D. Comba. Real-time multi-agent path planning on arbitrary surfaces. In *Proc. ACM SIGGRAPH I3D*, pages 47–54, 2010.

[14] S.J. Guy, J. Chhugani, C. Kim, N. Satish, M.C. Lin, M. Manocha, and P. Dubey. Clearpath: Highly collision avoidance for multi-agent simulation. In *Proc. ACM SIGGRAPH/EUROGRAPHICS SCA*, pages 177–187, 2009.

[15] A. Bleiweiss. Scalable multi agent simulation on the gpu. In *Proc. GPU Technology Conference*, 2009.

[16] N.R. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. In *Proc. AAAI*, pages 1392–1397, 2005.

[17] A. Pal, R. Tiwari, and A. Shukla. A Focused Wave Front Algorithm for Mobile Robot Path Planning. In *Proc. HAIS (1)*, pages 190–197, 2011.

[18] A. Hoisie, O.M. Lubeck, and H.J. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Proc. Wide Area Networks and High Performance Computing*, pages 171–187, 1998.

[19] S. Singh, M. Kapadia, B. Hewlett, G. Reinman, and P. Faloutsos. A modular framework for adaptive agent-based steering. In *Proc. ACM SIGGRAPH 13D*, pages 141–150, 2011.

[20] N. Sturtevant. Benchmarks for grid-based pathfinding. *Trans. on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.