A GPU-Based Concurrent Motion Planning Algorithm for 3D Euclidean Configuration Spaces

Stephen Cossell and José Guivant
School of Mechanical and Manufacturing Engineering,
The University of New South Wales, Australia
{macgyver,j.guivant}@unsw.edu.au

Abstract

For existing robot motion planning algorithms, it is well understood that the computational complexity of an implementation increases exponentially as the dimensionality of the configuration space increases. This paper presents an approach to 3D motion planning in Euclidean voxelised configuration spaces that runs in near linear time. It leverages the highly parallel architecture found in modern graphics hardware to expand many edges of a configuration space concurrently — a task traditional algorithms complete sequentially and in worse than linear time. With the continual increase in computation power available on modern parallel processing architectures, many motion planning problems that were once too computationally expensive for practical application can now be run in real-time.

1 Introduction

Motion planning is a crucial component in modern robotics systems and applies to platforms ranging from multiple link articulated robots to unmanned ground and aerial vehicles. While a precise path from an initial state to a goal state is often desired, a suboptimal solution, or a solution that uses less information about the environment, are often chosen to counter the negative effects of a lack of computational power available for real-time applications. Many solutions to this limitation have been presented, including reducing the resolution used to represent the configuration space, projecting the configuration space representation from $\mathbb{R}^p$ to $\mathbb{R}^q$ for $q < p$, and planning in $\mathbb{R}^{d-1}$ for an environment originally represented in $\mathbb{R}^d$ [KalinGochev *et al.*, 2011][Latombe, 1991]. Conversely, a common practice used to counter this lack of computational power over the last two decades was to upgrade to the next generation of processing unit. The advantage here was that the same algorithm could be run on the new processor without modification. In recent years, however, the single core processor has begun to reach its physical limit and manufacturers are no longer able to provide the generation to generation increase in computational power as they once did.

In contrast, multi-core central processing units (CPUs) and in particular highly parallel graphics processing units (GPUs) are becoming more prevalent and as such, many industry standard algorithms are being replaced by concurrent algorithms applicable to these parallel architectures. Basic data sorting [Satish *et al.*, 2009], pattern matching for applications in DNA sequencing [Trapnell and Schatz, 2009] and database operations in SQL [Bakkum and Skadron, 2010] are among some of the mainstream applications that have been proven to show an increase in performance on parallel architectures.

This paper presents a concurrent algorithm capable of generating a many to one cost-to-go function on a 3D voxel-based configuration space. Instead of running in approximately $\mathcal{O}(N^3)$ time for an $N \times N \times N$ volume, the concurrent algorithm performs closer to $\mathcal{O}(N)$, depending on the layout of obstacles in the configuration space. The technique extends upon a 2D algorithm presented in [Cossell and Guivant, 2011c] and later optimised in [Cossell and Guivant, 2011a] from real-time applications.

This paper begins by giving a survey of existing research that forms the basis of modern robot motion planning algorithms in two and three dimensions. The paper then continues to present the proposed algorithm before presenting a quantitative analysis. Section 5 briefly outlines current research investigating extending this algorithm to 4th and higher dimensional configuration spaces.

2 Prior Work

Motion planning algorithms have existed for a number of decades. Initial approaches tended to aim for mathematical optimality in the solutions the produced, but as these algorithms began to reach implementation, computational complexity became a significant factor in an algorithm's design. Dijkstra [Dijkstra, 1959] presented an initial approach which produced optimal results, but was found to be computationally expensive relative to less mathematically exact and more heuristic-based approaches, when applied to real world situations.

By combining a measurable heuristic into the core logic of Dijkstra's algorithm, Hart et al. [Hart *et al.*, 1968] showed that a mathematically optimal solution could be discovered in no worse time than Dijkstra's algorithm — with the notion that in many real world applications it provided a solution in less time. The key factor here was that, by including extra information about the configuration space's environment into the decision of which edge to explore next, the algorithm is able to potentially find the optimal solution in less steps.

While this class of algorithms provide an optimal solution in a given configuration space they are more often than not impractical for real-time, real-environment applications, due to the aforementioned computational complexity required to find a solution. Amato and Wu [Amato and Wu, 1996] noted that many practical applications in manipulator and vehicle planning resorted to probabilistic and heuristic methods over exact methods. They then went on to introduce a randomised road map method that appeared to solve some of the inaccuracies that come with low resolution and approximate methods. Although their approach appeared to generate paths in real-time, the generated paths were not optimal for many real-world applications.

One of the first papers presenting motion planning on modern graphics hardware highlighted that many preexisting approximate techniques, although capable of computing a solution for real-time applications, were poor at planning through detail dense parts of an environment due to representing the configuration space with too low resolution [Pisula *et al.*, 2000]. The paper introduced a graphics hardware-based implementation that used existing Voronoi diagram road map generation techniques, but could attempt to plan in greater detail when a narrow passage was detected from characteristics of the Voronoi diagram representation. While research such as this presented a more efficient method to plan paths through particularly detailed environments, it was still an approximate solution.

In [Cossell and Guivant, 2011c] the authors show the basic technique behind computing a complete and exhaustive cost-to-go function on a generated occupancy grid. The technique removed the mechanism of a priority queue of yet to be explored edges, and therefore the burden of calculating which edge to explore next. Instead, every unexplored edge that is adjacent to an explored edge in the grid-based environment is explored concurrently. In traditionally sequential algorithms such as Dijkstra's algorithm and its derivative, A* search, a node expands out along its unexplored edges and gives neighbouring nodes a cost value. In the concurrent version presented in [Cossell and Guivant, 2011c], a reversed direction of observation and calculation is applied. Each cell looks to its 8 neighbours in the grid and determines its value based on the lowest cost of a neighbouring cell combined with the cost of traveling between the two cells. The basic algorithm was later optimised for real world applications in [Cossell and Guivant, 2011a]. The method presented in this paper is a 3D extension of this 2D algorithm.

3 Concurrent Algorithm in 3D

The algorithm presented in this paper applies to configuration spaces that are represented by the approximate cell decomposition approach of voxelisation of the configuration space. That is, the configuration space is broken into equally sized cubes that tessellate across a volume containing the problem's environment. This maps well to the 2D and 3D based data structures offered by both OpenGL and OpenCL.

The volume representing the configuration space is initialised with cells in C-obstacle given a specific `OBSTACLE` value, the zero-cost destination cell given a value of 0, and all other cells in C-free are set to `UNDEFINED`. The analyses presented in this paper assume the agent occupies a single cell. By *growing* obstacles, as presented in [Lozano-Pérez and Wesley, 1979], a multiple cell agent can be represented as a single cell, while being capable of producing optimal results, to a resolution. The entire 3D buffer is then copied from CPU memory to GPU memory before computation is begun. The kernel given in 1 is then applied to each cell in the volume using a design pattern in GPGPU programming known as Ping-Ponging [Harris, 2005]. This involves alternating two 3D textures between the roles of read buffer and write buffer at each iteration. Due to the arbitrary evaluation order of kernels run concurrently, the Ping-Ponging method allows each memory block to be in a predictable state after each iteration of the algorithm, irrespective of the actual order cells were evaluated in during that iteration.

As the algorithm progresses, a virtual *wavefront* of cells are evaluated in three dimensions concurrently. An animation of this process can be see in [Cossell and Guivant, 2011b] with C-obstacle = $\emptyset$ and in [Cossell and Guivant, 2013] with a volume with selected axis aligned walls.

Algorithm 1 Kernel pseudocode run on each 3D voxel in the configuration space.

```
best cost := undefined
N := set of 26-way neighbours of current cell
for n ∈ N do
  if n_cost is defined then
    current cost := n_cost + travel_cost(n, current cell)
    if best cost is undefined or current cost is better
    than best cost then
      best cost := current cost
    end if
  end if
end for
current cell_cost := best cost
```

At present, the implementation determines when to terminate based on the zero maximum cell delta method presented in [Cossell and Guivant, 2011c]. The implementation is planned to incorporate the subregion approach in future development and hence will shift to the empty active list condition presented in [Cossell and Guivant, 2011a].

Once the algorithm has either exhaustively evaluated all cells in the configuration space, or evaluated cells up to and including the agent's location in the volume, the cell values within a given Manhattan radius of an agent's location are read back to CPU memory so that a short term path can be planned. The technique of gradient following [Koditschek, 1987] is used to discover the shortest path between the agent's location and the zero cost destination cell.

4 Algorithm Performance Analysis

Given a configuration space in $\mathbb{R}^3$ comprising $N \times N \times N$ cells, with C-obstacle = $\emptyset$. By setting the centre cell of this volume as the zero cost destination cell, a sequential algorithm will take $N^3 - 1$ steps to exhaustively evaluate the entire configuration space and hence run in $\mathcal{O}(N^3)$ time. The proposed concurrent algorithm is able to evaluate the entire volume in approximately $\frac{N}{2}$ iterations and hence, for this particular configuration space, runs in $\mathcal{O}(N)$ time. This configuration space layout demonstrates the theoretical best possible ratio of evaluated cells to number of iterations taken to achieve the exhaustive result.

Conversely, given a configuration space in $\mathbb{R}^3$ comprising $N \times N \times N$ cells, with the entire free space consisting of a single cell corridor path separated by obstacle cells. The zero cost destination cell is set as one of the cells at the end of the corridor. Here the proposed concurrent algorithm is given little to no opportunity to expand into more than one cell at any iteration and hence performs in the order of $\mathcal{O}(N^3)$, just like the sequential class of algorithms.

Configuration space representations used to approximate real-world environments sit somewhere between these two extremes. In particular, many environments have a favourable free space to obstacle space ratio and paths often involve routing around an obstacle rather than being forced for explore a contrived set of switchback or S-bend type routes[1]. Analysis of such properties of environments used during initial testing of the algorithm suggest the performance is capable of being closer to the $\mathcal{O}(N)$ end of the spectrum.

Removing a layer of abstraction on this representation, it should be noted that the evaluation of a single cell in the 2D version of the algorithm involves the comparison of 8 neighbouring cells in sequence, whereas the 3D version of the algorithm requires[2] 26. Factoring these differing sub-iteration kernel run times relative to each other, the effective number of cells that have been evaluated $|E|$ of the algorithm at sub-iteration j are given by Equations 1 and 2, respectively.

$$|E_{2D}| = (2 \times \left\lfloor \frac{j}{8} \right\rfloor + 1)^2 \tag{1}$$

$$|E_{3D}| = (2 \times \left\lfloor \frac{j}{26} \right\rfloor + 1)^3 \tag{2}$$

Here j is the unit time taken to compute and compare the value of one neighbouring cell in the kernel. One iteration of the 2D algorithm equates to multiple cells concurrently undergoing $8 \times j$ sub-iteration units of time, whereas the 3D algorithm will have consumed $26 \times j$ sub-iteration units of time. Generally, for any dimensionality $d \in \mathbb{Z}^+$, this equation can be expressed as Equation 3.

$$|E_d| = (2 \times \left\lfloor \frac{j}{3^d - 1} \right\rfloor + 1)^d \tag{3}$$

At the initial stages of an exhaustive evaluation, the 2D version of the algorithm will have been able to

[1] Examples include navigating a multi-storey building and outdoor environments used in submarine and aerial applications.

[2] The 2D version has a kernel consult $3^2 - 1$ neighbouring cells, whereas the 3D version consults $3^3 - 1$ neighbouring cells.

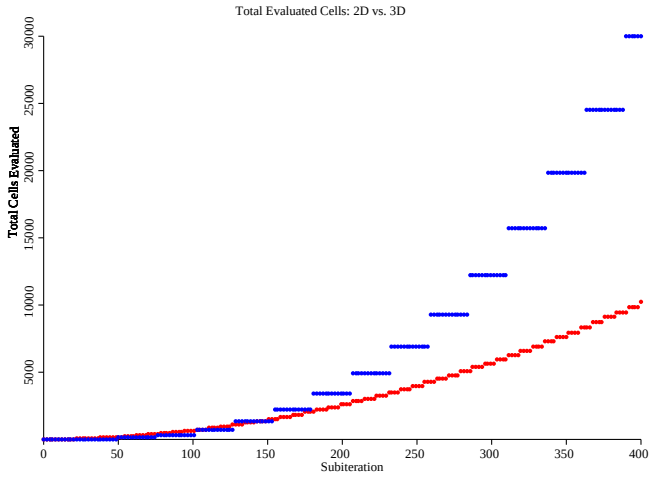

Figure 1: Comparison of the number of cells evaluated against the number of sub-iterations of the respective kernel programs of the 2D and 3D version of the algorithm. Blue marks represent the number of cells evaluated by the 3D version of the algorithm, with red representing those from the 2D algorithm.

advance 3 iterations (with a total of 24 sub-iterations worth of time) and evaluate up to $7^2 - 1$ cells before the 3D version of the algorithm is able to complete its first iteration (consisting of 26 sub-iterations and the evaluation of up to $3^3 - 1$ cells). That is, for the first 100 to 150 sub-iterations of evaluation, the 2D version of the algorithm is able to evaluate a greater number of cells than the 3D version of the algorithm. However, as the 3D algorithm expands into a larger number of cells each iteration, the concurrent benefit sees a significant benefit emerge. Figure 1 shows the potential number of cells that can be evaluated over the first 400 sub-iterations. Here 400 sub-iterations equates 50 iterations of the 2D version of the algorithm and 15 iterations of the 3D algorithm.

Theoretically, the 2D and 3D versions of the algorithm can evaluate an obstacle free space of s^2 and s^3 cells respectively in the same order of magnitude of iterations. Setting $s = 512$, the 2D version would evaluate in the order of 512^2 cells in approximately 2044 sub-iteration units of time, whereas the 3D version would evaluate in the order of 512^3 cells in approximately 6643 sub-iteration units of time.

Real-world occupancy grid representations have a favourable free space to obstacle space, with the *Help Points* occupancy grid given in [Cossell and Guivant, 2011a] having a 11.45% proportion of obstacle cells. Using the square-cube law, this ratio would approximate to 3.87%. for 3D occupancy volume. Generally, the increased concurrent benefit observed with an increase in dimensionality of the configuration space tends to counter the traditional rule of an exponential increase in the computational burden as the dimensionality increases.

In practice, this theoretical concurrent benefit is limited by the number of processors available. As the number of cells requested for evaluation each iteration (Q) gradually increases, these evaluations begin to queue up sequentially behind P processors, with $P < Q$. The transfer of data between CPU and GPU memory is also a significant limitation. However, unlike the physical limit single core processors have reached, these limiting factors are predicted to continue to improve for the foreseeable future [Borkar, 2007].

An initial implementation of the algorithm has been written in OpenCL. This language was chosen as it has particularly good support for 2D and 3D tessellated voxel-based data structures. Preliminary results are favourable relative to selected forefront sequential implementations. By adapting the subregion optimisation techniques presented in [Cossell and Guivant, 2011a] to the 3D implementation as subvolumes, results are predicted to be able show real-time evaluation for a 1024^3 cell configuration space.

5 Future Work

This paper presented a three-dimensional evolution of an existing two-dimensional algorithm. Further research is being conducted into higher dimensional implementations of this algorithm for applications in 4 and higher degree of freedom robot manipulators. In [Latombe, 1991], it is shown how an N degree-of-freedom robot manipulator's configuration space can be represented as a Euclidean space with axes representing the possible angles of each of the joints of the manipulator. Then, for the whole manipulator to be able to travel from point A to B, the evaluation and planning can be accomplished completely in angle-axis space as if the configuration space representation was in spatial-axis space.

As highlighted in Section 4, the concurrent benefit of the evaluation of a configuration space increases as the dimension of the configuration space increases. That is, for a 2D configuration space, the wavefront of evaluation at any increment of the algorithm is 1D. For a 3D configuration space, the wavefront expands as a

2D surface. Naturally, a 4D configuration space allows a 3D volume to expand concurrently. On current graphics hardware, this level of theoretical concurrency stars to hit diminishing returns as the number of concurrent calculations that are requested to be evaluated at any single iteration becomes a significantly larger than the number of cores available. However, the number of cores in a processor under a SIMD[3] architecture theoretically scales to an almost unlimited number of cores without any modification to the algorithm or implementation itself.

Work has been undertaken to begin implementing greater than 3D versions of the presented algorithm for robot manipulator and other optimisation problem applications. As OpenCL only supports up to three dimensions in its core data type buffers, a rescursive rastering method has been applied to these initial implementations to allow all required data to be available to the shader processors at each iteration. As such, a modified kernel has been implemented that is able to look up adjacent cell values in the 4th dimension using a linear function to work out where adjacent cells are stored in memory. Initial theoretical performance comparisons have shown no great burden in doing this initial linear look up compared to two and three dimensional adjacent cell value look ups.

6 Conclusion

This paper has presented a three dimensional extension to an existing two-dimensional concurrent dynamic programming technique. The algorithm has been shown theoretically to counter the accepted property of complete solutions to dynamic programming problems — that the computational complexity of generating a solution increases exponentially with the dimensionality of the configuration space. Preliminary experiments have shown favourable execution times, with further work currently being completed to alleviate specific implementation bottlenecks that have arisen. Many industrial applications of robot motion planning are relying on sequential algorithms implemented for an aging single core architecture. The class of algorithms presented in this paper enables scientists and engineers to harness the growing capabilities found in modern multi-core architectures.

References

[Amato and Wu, 1996] Nancy M. Amato and Yan Wu. A randomized roadmap method for path and manipulation planning. In *Proceedings of the 1996 IEEE Interational Conference on Robotics and Automation*, Minneapolis, Minnesota, USA, April 1996.

[Bakkum and Skadron, 2010] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, New York, NY, USA, 2010.

[Borkar, 2007] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, San Diego, California, USA, June 2007.

[Cossell and Guivant, 2011a] Stephen Cossell and José Guivant. Concurrent dynamic programming for grid-based problems and its application for real-time path planning. *Robotics and Autonomous Systems (under review revision)*, October 2011.

[Cossell and Guivant, 2011b] Stephen Cossell and José Guivant. GPGPU cost function generation for 3D occupancy volumes. `https://www.youtube.com/watch?v=1xEBLcis_94`, June 2011.

[Cossell and Guivant, 2011c] Stephen Cossell and José Guivant. Parallel evaluation of a spatial traversability cost function on gpu for efficient path planning. *Journal of Intelligent Learning Systems and Applications*, 3(4):191–200, November 2011.

[Cossell and Guivant, 2013] Stephen Cossell and José Guivant. GPGPU cost function generation on a non-empty 3D occupancy volume. `https://www.youtube.com/watch?v=t6-RMbk8mgE`, April 2013.

[Dijkstra, 1959] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[Harris, 2005] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH*, Los Angeles, California, USA, July 2005.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[KalinGochev *et al.*, 2011] KalinGochev, Benjamin Cohen, Jonathan Butzke, Alla Safonova, and Maxim Likhachev. Path planning with adaptive dimensionality. In *Fourth Annual Symposium on Combinitorial Search*, July 2011.

[Koditschek, 1987] D. E. Koditschek. Exact robot navigation by means of potential functions: Some topological considerations. In *Proceedings 1987 IEEE Conference on Robotics and Automation*, March 1987.

[3]SIMD stands for Single Instruction, Multiple Data, and describes the same small portion of code, or kernel, being executed on each element in an OpenGL or OpenCL buffer.

[Latombe, 1991] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Lozano-Pérez and Wesley, 1979] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, October 1979.

[Pisula *et al.*, 2000] Charles Pisula, K. Hoff, Ming Lin, and Dinesh Manocha. Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In *Proc. Workshop on Algorithmic Foundation of Robotics*, volume 18, 2000.

[Satish *et al.*, 2009] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel and Distributed Processing*, Rome, May 2009.

[Trapnell and Schatz, 2009] Cole Trapnell and Michael C. Schatz. Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing*, 35(8-9):429–440, August-September 2009.