The Secrets of Parallel Pathfinding on Modern Computer Hardware

by Jad Nohra and Alex J. Champandard

Credits: The A* implementation used for the experiments in this article was written by Jad Nohra and Nick Samarin. The parallel query processor and job scheduler were written by Alex J. Champandard and Radu Septimiu Cristea on top of Intel® Threading Building Blocks (Intel® TBB) technology.

Pathfinding in games is often very high on the list of expensive operations and is therefore one of the first things that artificial intelligence (AI) developers parallelize. But the most common approach is to fire off the pathfinder in a separate thread and stop worrying about it. How effective is this? Does it play well on modern computer processors? What's going on under the hood?

In the past year, the games industry has turned its focus to memory efficiency and "data-oriented" design of software [1]—in particular, to improve performance on multi-core hardware. The idea is that you should think more about your data-structures to reduce cache misses, which are often the biggest sources of inefficiencies.

What does this data-oriented approach mean for pathfinding? Do you need to be particularly careful when designing your graph data structures, or is the A* algorithm somehow immune to underlying memory accesses? More importantly, how does this affect parallel code? Is it effective to throw more processors at the problem, or should pathfinding calculations be run in a serial order on their own core?

Best Case Parallel Performance

To dig into the problem of parallel pathfinding, this article uses the A* algorithm from the AI Sandbox [2]. The implementation uses template-based techniques and shares many characteristics with production code—in particular, Relic's A* implementation released by Chris Jurney [3]. The underlying representation used is a 2D grid stored in an array-like data structure, similar to many real-time strategy games.

In practice, the AI Sandbox has multiple mini-games and uses the following approach to enable parallel pathfinding for its AI actors:

- Agent updates run sequentially in a first (pre-update) phase and send all their queries to a central query manager built on top of Intel® TBB [4].
- In the following phase, the manager has gathered the workload, creates jobs for the queries, and executes them all in parallel.
- Finally, in a third (post-update) phase, the agents are notified and can act based on the query results.

Despite the typical overheads of this batch-centered design [5], this implementation can be used to provide some accurate insights into theoretical A* pathfinding performance and its scalability when only constrained by hardware bottlenecks and low-level software overhead (for example, Intel® TBB).

To measure this, we wrote a test that performs the following tasks:

- Generates a large number of pseudo-random pathfinding queries to execute on the same map
- Runs in a tight loop, batching queries based on a batch size parameter
- Processes the batches in parallel, repeating the process until all queries have been processed

During the tests, each run had to execute a total of 1000 queries on a search graph of 1600 nodes occupying around 64 KB in memory. Controlled random seeds were used for maps and queries, effectively producing the exact same "randomness" on every run.

Multiple hardware configurations were tested, and the number of hardware threads was controlled from code using the operating system's process affinity functions, which forces processors with hyperthreading to use a minimal number of physical cores. Multiple runs were executed for each configuration, with batch sizes increasing from one to the maximum possible. For each run, only time spent pathfinding and inside parallel code was measured (with microsecond accuracy) and summed up to give the total effective time of the run. Figure 1 shows the results.

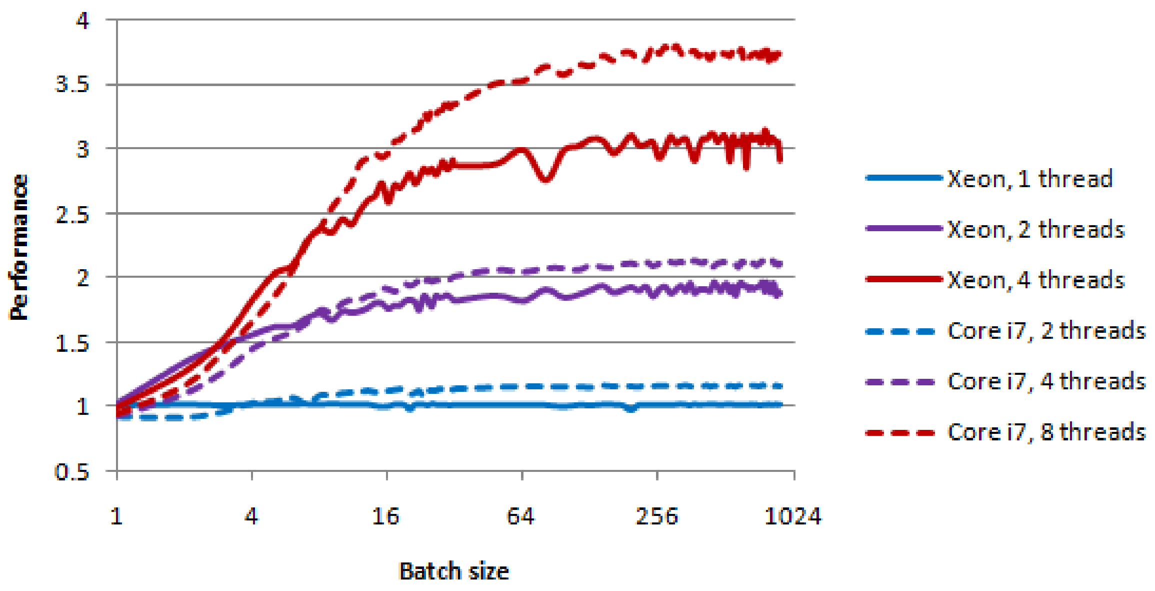


Figure 1. Relative performance with additional cores while increasing batch size The performance of the test using one Intel® Xeon® processor core was taken as a baseline for the measuring performance. The test was run on two processors: (1) an Intel® Xeon® processor 5460, Intel® Core™ microarchitecture, 3.16 GHz, four cores, four hardware threads, a 4×32 KB L1 D-cache, a 4×32 KB L1 I-cache, and a 2×6 MB L2 cache and (2) an Intel® Core™ i7 processor 920, Intel® microarchitecture codename Nehalem, 2.66 GHz, four cores, eight hardware

threads, a 4×32 KB L1 D-cache, a 4×32 KB L1 I-cache, a 4×256 KB L2 cache, and a 1×8 MB L3 cache.

Both processors manage approximately double performance with two physical cores. The Intel® Xeon® processor maxed out at 3× speed-up, while the Intel® Core™ i7 processor managed to reach the 3.75 mark with its eight hyperthreads. As expected, large batch sizes meant less overhead and better scaling. In the case of this specific test, the overhead became negligible at around 256 queries per batch.

This first high-level test shows the kind of scalability possible using parallelism under near-ideal conditions, but we only used one fixed map. How does this turn out in practice on real-world maps?

Performance on Different Map Types

Although many performance subtleties are related to genre or game-specific maps and query patterns [6], this section digs into map size and difficulty variations in particular, because these are commonly known factors that affect single-threaded pathfinding performance.

Experimenting with map size is interesting for at least two reasons:

- Given common wisdom about cache performance, we expect map size to have an obvious effect on performance because of memory access stalls.
- Because modern processors have very different microarchitectures, we're curious about the relative performance of both processors with their noticeably different cache designs.

To test its effect, we fixed the batch size to 64 and ran the test over maps of varying sizes. Figure 2 and Figure 3 show the same results, presented as different statistics.

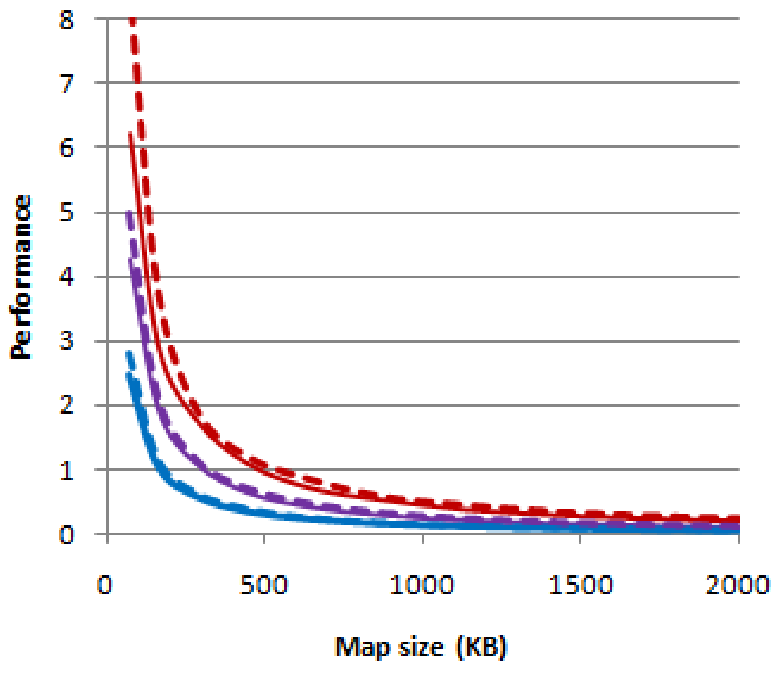


Figure 2. Performance degrades incrementally and continuously as maps grow larger but shows no clear drops at L1 D-cache size boundaries. The degradation is similar for both processors despite their differences, which will require further investigation in the next section.

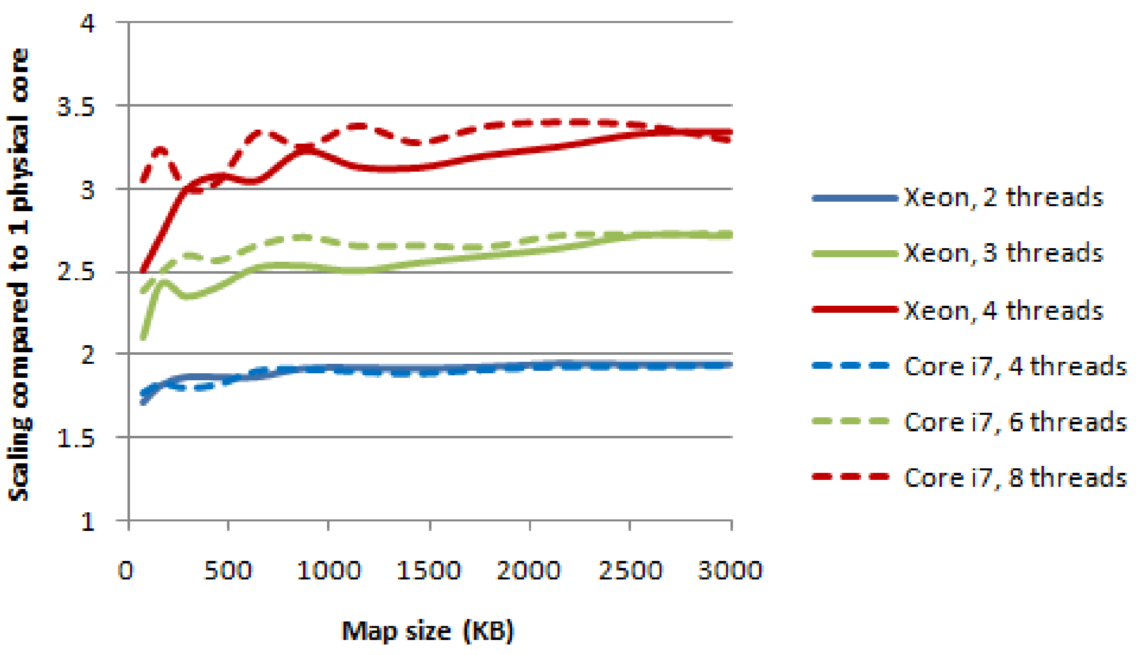


Figure 3. Speed-up mostly remains stable and unaffected by map size. We speculate that the fluctuations for very small maps sizes are related to the increase of measurement inaccuracy (smaller maps have very short queries that execute extremely quickly), and in these scenarios,

the constant overhead of the job queue plays a bigger role. If anything, this discrepancy emphasizes the need for profiling on a case-by-case basis, especially at extremes (such as very small maps).

Second, to simulate various map difficulties, we added a parameter that varies the maximum magnitude of the random costs assigned to the search graph edges. As these costs increase, the heuristic suffers and A* needs to work harder by searching a much larger space—a known performance problem for single-threaded pathfinding. In these experiments, we fixed the batch and map size and ran several tests while increasing difficulty. Figure 4 shows the results.

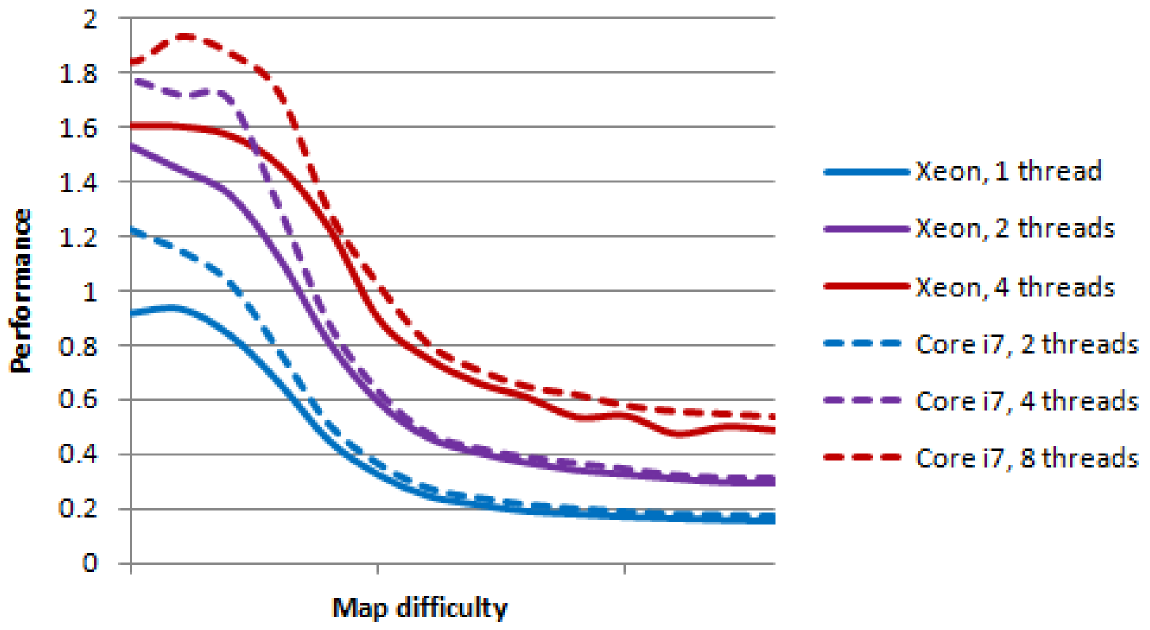


Figure 4. Mu**lti-core performance with increasing map difficulty** Again, performance degraded with difficulty, and speed-up remained mostly stable. The slightly irregular curves indicate that, unlike in previous tests, each run had to use different random edge costs, so maps with slightly larger difficulty settings could still on average be “easier” than those with slightly lower costs.

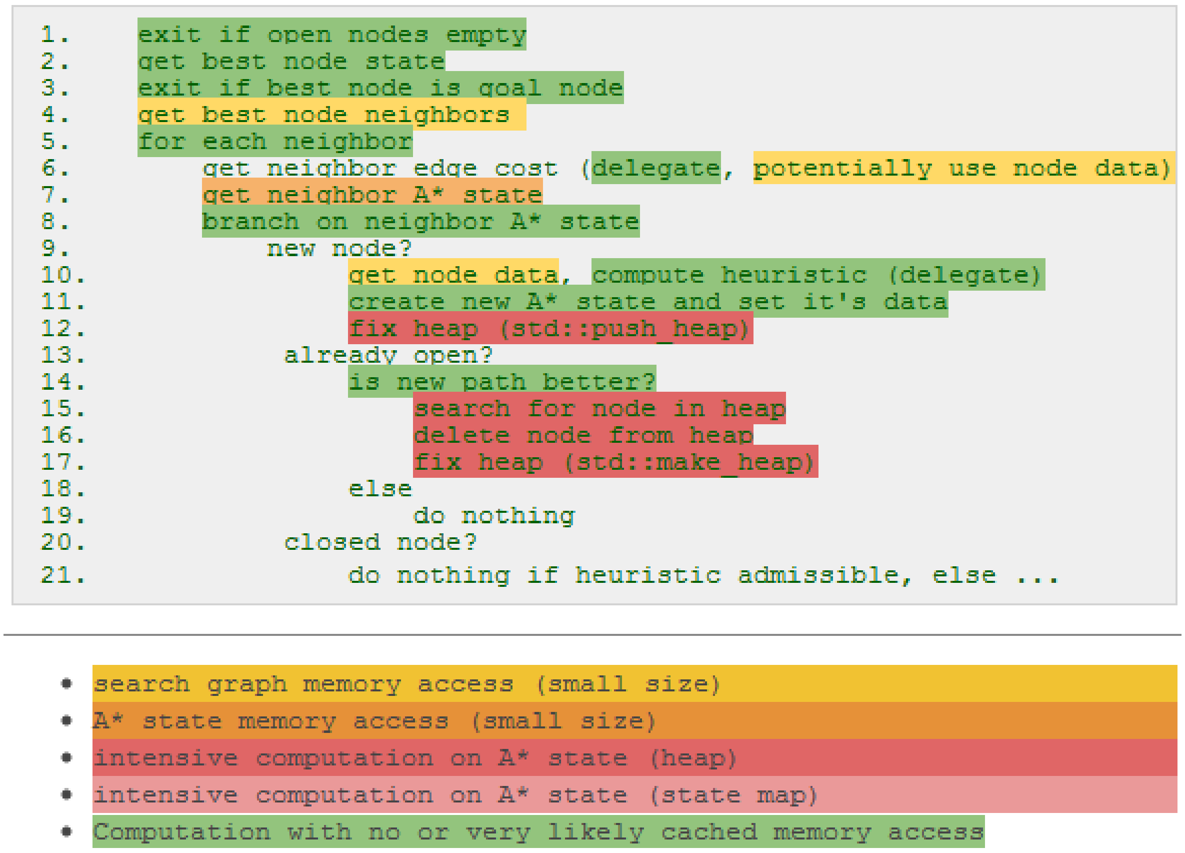
These two experiments show that parallel pathfinding performance suffers by varying map features, similar to single threads. However, it’s interesting to see that multi-core scaling generally remains unaffected by map size or difficulty. Digging into the code and its behavior should help explain why this is the case.

Investigating Memory Footprint

Memory footprint is the natural factor to look at when it comes to map size increases. As the aggregate size in the memory of the search graph and the search query’s A* structures grows, increased numbers of cache misses can lead to memory access stalls and degrade performance. Is this the main reason behind performance degradation with map size in the previous tests?

To answer this question, a more thorough test was needed rather than simply increasing map size. The test needs to use the same maps with different memory footprints, but without increasing the actual graph structure, which would otherwise reflect the performance impact of different query lengths. In practice, we created a test that varies memory footprint alone—without affecting any other factors—by artificially manipulating the size of underlying data-structures. This was achieved by padding nodes in the graph with up to 64 additional vectors, increasing the node size from 12 bytes to 768 bytes. Although the padding vectors were not used in the heuristic, they increased not only the count of compulsory and conflict cache misses, but also capacity misses due to cache line granularity.

We started with a map that had an aggregate size that would fit in the L1 D-cache of both processors for 12 bytes per node and increased the size to 500 KB. All runs exhibited remarkably similar performance on both processors and showed that memory footprint had no effect. This result is quite surprising, but a short look at modern processor designs [7] revealed how this was possible. Modern processors excel at instruction-level parallelism (ILP) and can, under favorable circumstances, completely hide cache latency by making sure that cache misses do not stall the processor. If we examine the A* algorithm depicted in Figure 5, we notice that its structure does indeed allow this to happen.



```
exit if open nodes empty
get best node state
exit if best node is goal node
get best node neighbors
for each neighbor
    get neighbor edge cost (delegate, potentially use node data)
    get neighbor A* state
    branch on neighbor A* state
        new node?
            get node data, compute heuristic (delegate)
            create new A* state and set it's data
            fix heap (std::push_heap)
          already open?
            is new path better?
                search for node in heap
                delete node from heap
                fix heap (std::make_heap)
            else
                do nothing
          closed node?
            do nothing if heuristic admissible, else ...
```

- 🟨 search graph memory access (small size)
- 🟧 A* state memory access (small size)
- 🟥 intensive computation on A* state (heap)
- 🟥 intensive computation on A* state (state map)
- 🟩 Computation with no or very likely cached memory access

Figure 5. Sandbox A* implementation data access patterns

The accesses to search graph and mapping structure data are punctual and surrounded by a decent amount of processing, which works well with speculative execution (a standard feature since the P6 microarchitecture introduced in 1995) and allows a processor to execute instructions that lie beyond the conditional branches that depend on the accessed values. To give a few concrete numbers, even the older P6 microarchitecture has a deep out-of-order execution pipeline that supports 126 instructions in flight and up to 48 loads and 24 (or more) stores in pipeline [8].

The Intel® Core™ microarchitecture (introduced in 2006) also added advanced hardware prefetching. Although it is possible for this feature to worsen performance in rare cases, it clearly plays well with A*. When a node is accessed, its horizontal neighbors are close and its vertical neighbors are at a fixed stride distance—both cases can be predicted and prefetched. Also, an A* search will be working around already visited nodes most of the time. The Intel® microarchitecture codename Nehalem improved recovery from branch misprediction, and its cache latencies are impressive. For the Intel® Core™ i7 processor, they are four cycles for the L1 cache and only 10 cycles for the L2 cache; the L3 cache has a higher latency at more than 35 cycles.

All these ILP features are present in current multi-core processors; with the low latencies and deep pipelines, they seem to be able to completely hide cache latency for A* on common map sizes. For the Intel® Xeon® processor with its large L2 cache, it is easy to see how this can happen, but the Intel® Core™ i7 processor manages the same feat despite its much smaller L2 cache and its relatively high L3 latency, which is impressive and good news for AI on PC hardware!

Priority Queue Operations

Having established that memory footprint alone does not account for much performance degradation, we turned our attention to the next suspect: priority queue operations. These operations are the heartbeat of the A* algorithm, and most of the intense work happens there. For each visited search graph node, depending on whether it is new or already open, one of two different and expensive operations has to be executed: either opening an unvisited node or updating an existing one.

The AI Sandbox A* implementation uses a binary heap, which is a common choice. This representation is characterized by faster performance when opening new nodes than when updating open nodes [9]. Also, the time it takes for either one of the heap operations to execute depends heavily on the order in which nodes were visited and varies per query.

To find out more, we wrote a test that measures the average time taken to execute queries of a given length and also counts the number of operations of each type. The test ran over various maps of different sizes and difficulty (see Figure 6).

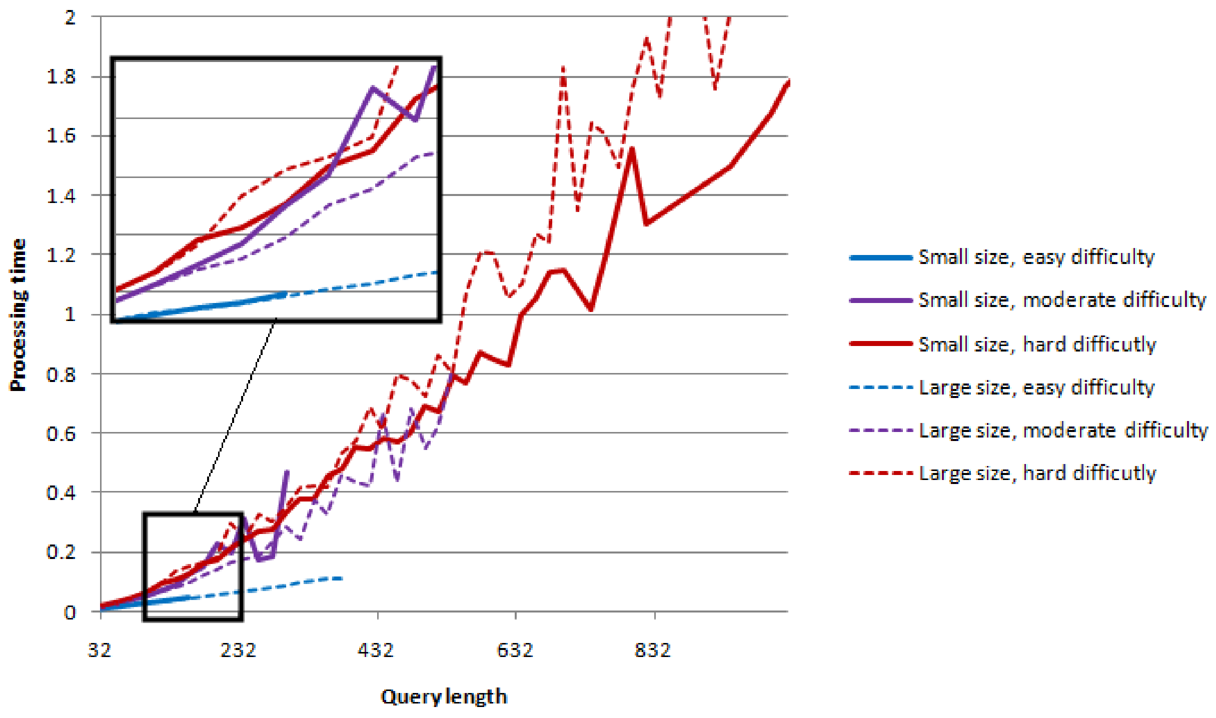


Figure 6. Performance relative to query length on various maps

As expected, performance degraded with query length and again, map size had no noticeable effect. Increased difficulty caused queries of the same length to execute more slowly. By looking at the operation counts (not included in the graph), we noticed that this slow-down was the result of a higher number of the slower operations, the frequency of which increases with map difficulty, as nodes have to be updated more often. This finding also explained the bumpiness of the graph, because query length is a rough number related to a number of executions of both operations, with each operation having a running time that varies depending on the exact state of the binary heap at that point.

Summary

Generally speaking—and perhaps surprisingly—you can throw more processors and threads at the A* pathfinding problem and it will scale better than most problems. The heart of the algorithm is so processor intensive that the overheads of parallelization remain insignificant. Better still, we found that modern processors such as the Intel® Xeon® processor or Intel® Core™ i7 processor manage to hide cache access latencies in our implementation on all map sizes that fit in the highest level of the cache. This is the case even for single queries, but using batched queries is more effective in parallel.

In practice—and this lesson applies to any type of hardware—trying to squeeze a map into a smaller memory size won't provide as much of a performance improvement compared to reducing the size of the graphs (for example, partitioning the world) or decreasing their complexity (for example, by multiplying the Euclidian heuristic by a constant factor). This is also good news for multi-threading, as this advice remains the same regardless of whether you're designing your pathfinder for parallel execution.

The tests in the first part of this article also hinted at the benefits of large batch sizes, which reduce the overheads of the job system as well as inefficiencies in scheduling. In the future, we plan to investigate the impact of having different graph representations (for example, for different agents) and multiple batches as well as the cost of interleaving or co-scheduling pathfinding with other types of operations, such as line-of-sight queries [4,5].

References

[1] Tony Albrecht, The Latency Elephant.

[2] The AI Sandbox, http://aisandbox.com.

[3] Chris Jurney, Relic's Templatized A* Implementation.

[4] Alex J. Champandard, Multi-threading Line-of-Sight Calculations to Improve Sensory System Performance in Game AI.

[5] Alex J. Champandard, AI Reasoning and Workload Management of Parallel Sensor Queries in Games.

[6] AI Wisdom (A*).

[7] Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[8] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1.

[9] Amit Patel, Pathfinding.