Retrofitted Parallelism Considered Grossly Sub-Optimal

Paul E. McKenney
Linux Technology Center
IBM Beaverton

Abstract

Maze solving has been used as an example parallel-programming problem for some years. Suggested solutions are often based on a sequential program, using work queues to allow multiple threads to explore different portions of the maze concurrently. This paper analyzes such an implementation, but also explores an alternative implementation based on strategies long used by human maze solvers. This alternative implementation outperforms the conventional approach on average, and furthermore exhibits large superlinear speedups. The paper uses insights into the cause of these superlinear speedups to derive a faster sequential algorithm, and finally considers further implications and future work.

1 Introduction

Labyrinths and mazes have been objects of fascination for millenia [15], so it should come as no surprise that they are generated and solved using computers, including biological computers [1], GPGPUs [6], and even discrete hardware [10]. Parallel solution of mazes is sometimes used as a class project in universities [7, 14] and as a vehicle to demonstrate the benefits of parallel-programming frameworks [8].

Common advice is to use a parallel work-queue algorithm (PWQ) [7, 8]. This paper evaluates this advice by comparing PWQ against a sequential algorithm (SEQ) and also against an alternative parallel algorithm, in all cases solving randomly generated square mazes. Section 2 discusses PWQ, Section 3 discusses an alternative parallel algorithm, Section 4 analyzes its anomalous performance, Section 5 derives an improved sequential algorithm from the alternative parallel algorithm, Section 6 makes further performance comparisons, and finally Section 7 presents future directions and concluding remarks.

```
 1 int maze_solve(maze *mp, cell sc, cell ec)
 2 {
 3   cell c = sc;
 4   cell n;
 5   int vi = 0;
 6
 7   maze_try_visit_cell(mp, c, c, &n, 1);
 8   for (;;) {
 9     while (!maze_find_any_next_cell(mp, c, &n)) {
10       if (++vi >= mp->vi)
11         return 0;
12       c = mp->visited[vi].c;
13     }
14     do {
15       if (n == ec) {
16         return 1;
17       }
18       c = n;
19     } while (maze_find_any_next_cell(mp, c, &n));
20     c = mp->visited[vi].c;
21   }
22 }
```

Figure 1: SEQ Pseudocode

2 Work-Queue Parallel Maze Solver

PWQ is based on SEQ, which is shown in Figure 1. The maze is represented by a 2D array of cells and a linear-array-based work queue named `->visited`.

Line 7 visits the initial cell, and each iteration of the loop spanning lines 8-21 traverses passages headed by one cell. The loop spanning lines 9-13 scans the `->visited[]` array for a visited cell with an unvisited neighbor, and the loop spanning lines 14-19 traverses one fork of the submaze headed by that neighbor. Line 20 initializes for the next pass through the outer loop.

The pseudocode for `maze_try_visit_cell()` is shown on lines 1-12 of Figure 2. Line 4 checks to see if cells `c` and `n` are adjacent and connected, while line 5 checks to see if cell `n` has not yet been visited. The `celladdr()` function returns the address of the specified cell. If either check fails, line 6 returns failure. Line 7 indicates the next cell, line 8 records this cell in the next slot of the `->visited[]` array, line 9 indicates that this

```
 1 int maze_try_visit_cell(struct maze *mp, cell c, cell t,
 2                         cell *n, int d)
 3 {
 4   if (!maze_cells_connected(mp, c, t) ||
 5       (*celladdr(mp, t) & VISITED))
 6     return 0;
 7   *n = t;
 8   mp->visited[mp->vi] = t;
 9   mp->vi++;
10   *celladdr(mp, t) |= VISITED | d;
11   return 1;
12 }
13
14 int maze_find_any_next_cell(struct maze *mp, cell c,
15                             cell *n)
16 {
17   int d = (*celladdr(mp, c) & DISTANCE) + 1;
18
19   if (maze_try_visit_cell(mp, c, prevcol(c), n, d))
20     return 1;
21   if (maze_try_visit_cell(mp, c, nextcol(c), n, d))
22     return 1;
23   if (maze_try_visit_cell(mp, c, prevrow(c), n, d))
24     return 1;
25   if (maze_try_visit_cell(mp, c, nextrow(c), n, d))
26     return 1;
27   return 0;
28 }
```

Figure 2: SEQ Helper Pseudocode

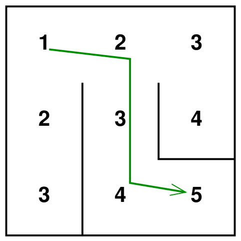

Figure 3: Cell-Number Solution Tracking

slot is now full, and line 10 marks this cell as visited and also records the distance from the maze start. Line 11 then returns success.

The pseudocode for maze_find_any_next_cell() is shown on lines 14-28 of the figure. Line 17 picks up the current cell's distance plus 1, while lines 19, 21, 23, and 25 check the cell in each direction, and lines 20, 22, 24, and 26 return true if the corresponding cell is a candidate next cell. The prevcol(), nextcol(), prevrow(), and nextrow() each do the specified array-index-conversion operation. If none of the cells is a candidate, line 27 returns false.

The path is recorded in the maze by counting the number of cells from the starting point, as shown in Figure 3, where the starting cell is in the upper left and the ending cell is in the lower right. Starting at the ending cell and following consecutively decreasing cell numbers traverses the solution.

The parallel work-queue solver is a straightforward parallelization of the algorithm shown in Figures 1 and

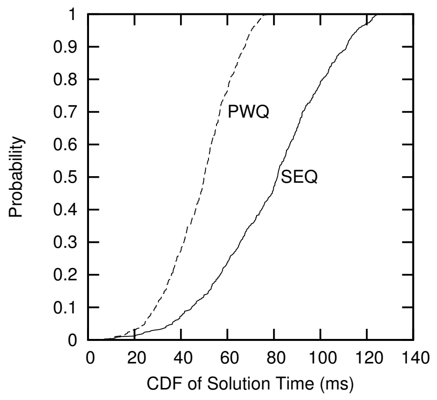

Figure 4: CDF of Solution Times For SEQ and PWQ

2. Line 10 of Figure 1 must use fetch-and-add, and the local variable vi must be shared among the various threads. Lines 5 and 10 of Figure 2 must be combined into a CAS loop, with CAS failure indicating a loop in the maze. Lines 8-9 of this figure must use fetch-and-add to arbitrate concurrent attempts to record cells in the ->visited[] array.

This approach does provide significant speedups on a dual-CPU Lenovo™ W500 running at 2.53GHz, as shown in Figure 4, which shows the cumulative distribution functions (CDFs) for the solution times of the two algorithms, based on the solution of 500 different square 500-by-500 randomly generated mazes. The substantial overlap of the projection of the CDFs onto the x-axis will be addressed in Section 4.

Interestingly enough, the sequential solution-path tracking works unchanged for the parallel algorithm. However, this uncovers a significant weakness in the parallel algorithm: At most one thread may be making progress along the solution path at any given time. This weakness is addressed in the next section.

3 Alternative Parallel Maze Solver

Youthful maze solvers are often urged to start at both ends, and this advice has been repeated more recently in the context of automated maze solving [14]. This advice amounts to partitioning, which has been a powerful parallelization strategy in the context of parallel programming for both operating-system kernels [3, 9] and applications [13]. This section applies this strategy, using two child threads that start at opposite ends of the solution path, and takes a brief look at the performance and scalability consequences.

The partitioned parallel algorithm (PART), shown in Figure 5, is similar to SEQ, but has a few important differences. First, each child thread has its own visited

```
 1 int maze_solve_child(maze *mp, cell *visited, cell sc)
 2 {
 3   cell c;
 4   cell n;
 5   int vi = 0;
 6
 7   myvisited = visited; myvi = &vi;
 8   c = visited[vi];
 9   do {
10     while (!maze_find_any_next_cell(mp, c, &n)) {
11       if (visited[++vi].row < 0)
12         return 0;
13       if (ACCESS_ONCE(mp->done))
14         return 1;
15       c = visited[vi];
16     }
17     do {
18       if (ACCESS_ONCE(mp->done))
19         return 1;
20       c = n;
21     } while (maze_find_any_next_cell(mp, c, &n));
22     c = visited[vi];
23   } while (!ACCESS_ONCE(mp->done));
24   return 1;
25 }
```

Figure 5: Partitioned Parallel Solver Pseudocode

array, passed in by the parent as shown on line 1, which must be initialized to all [-1,-1]. Line 7 stores a pointer to this array into the per-thread variable `myvisited` to allow access by helper functions, and similarly stores a pointer to the local visit index. Second, the parent visits the first cell on each child's behalf, which the child retrieves on line 8. Third, the maze is solved as soon as one child locates a cell that has been visited by the other child. When `maze_try_visit_cell()` detects this, it sets a `->done` field in the maze structure. Fourth, each child must therefore periodically check the `->done` field, as shown on lines 13, 18, and 23. The `ACCESS_ONCE()` primitive must disable any compiler optimizations that might combine consecutive loads or that might reload the value. A C++1x volatile relaxed load suffices [4]. Finally, the `maze_find_any_next_cell()` function must use compare-and-swap to mark a cell as visited, however no constraints on ordering are required beyond those provided by thread creation and join.

The pseudocode for `maze_find_any_next_cell()` is identical to that shown in Figure 2, but the pseudocode for `maze_try_visit_cell()` differs, and is shown in Figure 6. Lines 8-9 check to see if the cells are connected, returning failure if not. The loop spanning lines 11-18 attempts to mark the new cell visited. Line 13 checks to see if it has already been visited, in which case line 16 returns failure, but only after line 14 checks to see if we have encountered the other thread, in which case line 15 indicates that the solution has been located. Line 19 updates to the new cell, lines 20 and 21 update this thread's visited array, and line 22 returns success.

Performance testing revealed a surprising anomaly, shown in Figure 7. The median solution time for PART

```
 1 int maze_try_visit_cell(struct maze *mp, int c, int t,
 2       int *n, int d)
 3 {
 4   cell_t t;
 5   cell_t *tp;
 6   int vi;
 7
 8   if (!maze_cells_connected(mp, c, t))
 9     return 0;
10   tp = celladdr(mp, t);
11   do {
12     t = ACCESS_ONCE(*tp);
13     if (t & VISITED) {
14       if ((t & TID) != mytid)
15         mp->done = 1;
16       return 0;
17     }
18   } while (!CAS(tp, t, t | VISITED | myid | d));
19   *n = t;
20   vi = (*myvi)++;
21   myvisited[vi] = t;
22   return 1;
23 }
```

Figure 6: Partitioned Parallel Helper Pseudocode

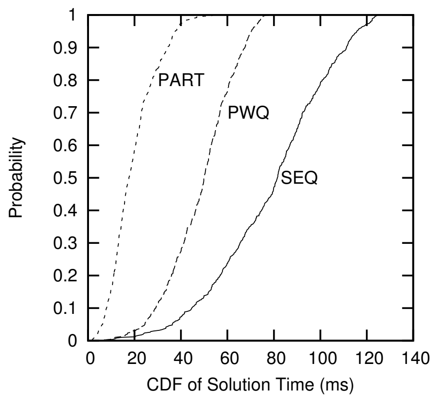

Figure 7: CDF of Solution Times For SEQ, PWQ, and PART

(17 milliseconds) is more than four times faster than that of SEQ (79 milliseconds), despite running on only two threads. The next section analyzes this anomaly.

4 Performance Comparison I

The first reaction to a performance anomaly is to check for bugs. Although the algorithms were in fact finding valid solutions, the plot of CDFs in Figure 7 assumes independent data points. This is not the case: The performance tests randomly generate a maze, and then run all solvers on that maze. It therefore makes sense to plot the CDF of the ratios of solution times for each generated maze, as shown in Figure 8, greatly reducing the CDFs' overlap. This plot reveals that for some mazes, PART is more than *forty* times faster than SEQ. In contrast, PWQ is never more than about two times faster than SEQ. A

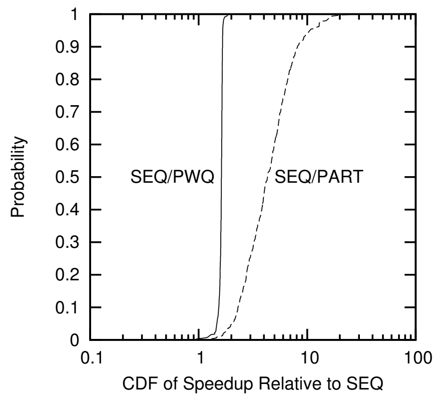

Figure 8: CDF of SEQ/PWQ and SEQ/PART Solution-Time Ratios

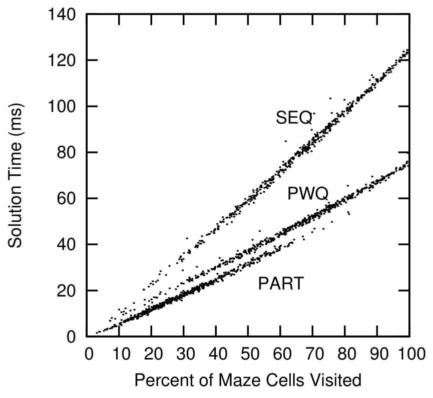

Figure 10: Correlation Between Visit Percentage and Solution Time

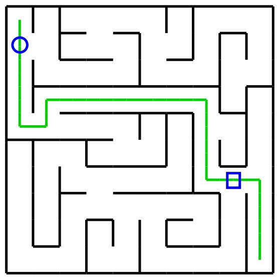

Figure 9: Reason for Small Visit Percentages

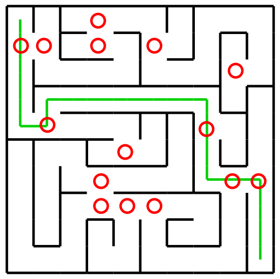

Figure 11: PWQ Potential Contention Points

forty-times speedup on two threads demands explanation. After all, this is not merely embarrassingly parallel, where partitionability means that adding threads does not increase the overall computational cost. It is instead *humiliatingly parallel*: Adding threads significantly reduces the overall computational cost, resulting in large algorithmic superlinear speedups.

Further investigation showed that PART sometimes visited fewer than 2% of the maze's cells, while SEQ and PWQ never visited fewer than about 9%. The reason for this difference is shown by Figure 9. If the thread traversing the solution from the upper left reaches the circle, the other thread cannot reach the upper-right portion of the maze. Similarly, if the other thread reaches the square, the first thread cannot reach the lower-left portion of the maze. Therefore, PART will likely visit a small fraction of the non-solution-path cells. In short, the superlinear speedups are due to threads getting in each others' way. This is a sharp contrast with decades of experience with parallel programming, where workers have struggled to keep threads *out* of each others' way.

Figure 10 confirms a strong correlation between cells visited and solution time for all three methods. The slope of PART's scatterplot is smaller than that of SEQ, indicating that PART's pair of threads visits a given fraction of the maze faster than can SEQ's single thread. PART's scatterplot is also weighted toward small visit percentages, confirming that PART does less total work, hence the observed humiliating parallelism.

The fraction of cells visited by PWQ is similar to that of SEQ. In addition, PWQ's solution time is greater than that of PART, even for equal visit fractions. The reason for this is shown in Figure 11, which has a red circle on each cell with more than two neighbors. Each such cell can result in contention in PWQ, because one thread can enter but two threads can exit, which hurts performance [11]. In contrast, PART can incur such contention but once, namely when the solution is located. Of course, SEQ never contends.

Although PART's speedup is impressive, we should not neglect sequential optimizations. Figure 12 shows that SEQ, when compiled with -O3, is about twice as fast as unoptimized PWQ, approaching the performance of unoptimized PART. Compiling all three algorithms with -O3 gives results similar to (albeit faster than) those shown in Figure 8, except that PWQ provides almost no speedup compared to SEQ, in keeping with Amdahl's Law [2]. However, if the goal is to double performance

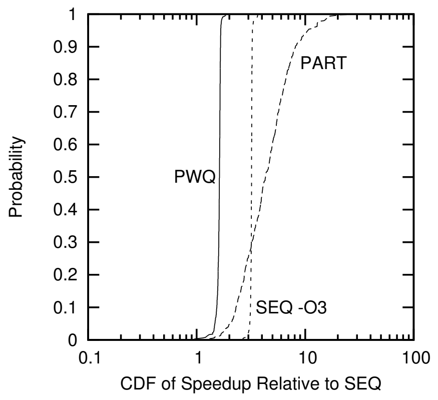

Figure 12: Effect of Compiler Optimization (-O3)

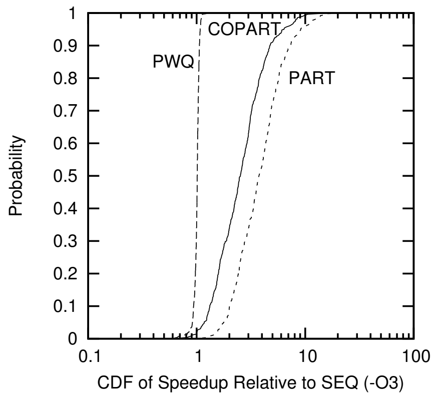

Figure 13: Partitioned Coroutines

compared to unoptimized SEQ, as opposed to achieving optimality, compiler optimizations are quite attractive.

Cache alignment and padding often improves performance by reducing false sharing. However, for these maze-solution algorithms, aligning and padding the maze-cell array *degrades* performance by up to 42% for 1000x1000 mazes. Cache locality is more important than avoiding false sharing, especially for large mazes. For smaller 20-by-20 or 50-by-50 mazes, aligning and padding can produce up to a 40% performance improvement for PART, but for these small sizes, SEQ performs better anyway because there is insufficient time for PART to make up for the overhead of thread creation and destruction.

In short, the partitioned parallel maze solver is an interesting example of an algorithmic superlinear speedup. If "algorithmic superlinear speedup" causes cognitive dissonance, please proceed to the next section.

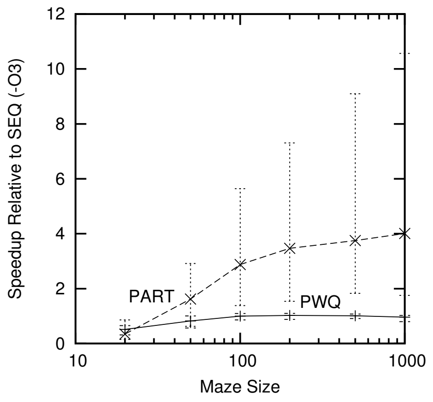

Figure 14: Varying Maze Size vs. SEQ

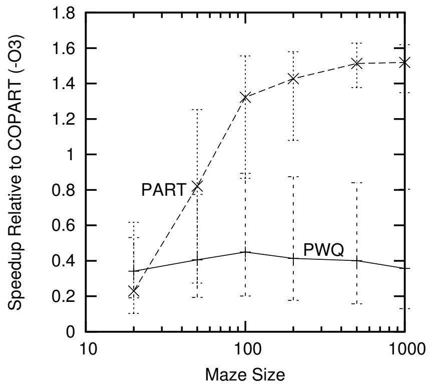

Figure 15: Varying Maze Size vs. COPART

5 Alternative Sequential Maze Solver

The presence of algorithmic superlinear speedups suggests simulating parallelism via co-routines, for example, manually switching context between threads on each pass through the main do-while loop in Figure 5. This context switching is straightforward because the context consists only of the variables `c` and `vi`: Of the numerous ways to achieve the effect, this is a good tradeoff between context-switch overhead and visit percentage. As can be seen in Figure 13, this coroutine algorithm (COPART) is quite effective, with the performance on one thread being within about 30% of PART on two threads.

6 Performance Comparison II

Figures 14 and 15 show the effects of varying maze size, comparing both PWQ and PART running on two threads against either SEQ or COPART, respectively, with 90%-confidence error bars. PART shows superlinear scalability against SEQ and modest scalability against COPART for 100-by-100 and larger mazes. PART exceeds the-

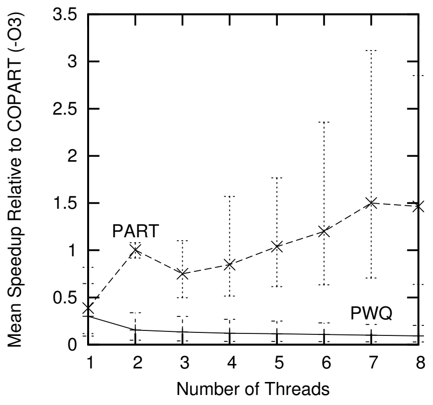

Figure 16: Mean Speedup vs. Number of Threads, 1000x1000 Maze

oretical energy-efficiency breakeven against COPART at roughly the 200-by-200 maze size, given that power consumption rises as roughly the square of the frequency for high frequencies [12], so that 1.4x scaling on two threads consumes the same energy as a single thread at equal solution speeds. In contrast, PWQ shows poor scalability against both SEQ and COPART unless unoptimized: Figures 14 and 15 were generated using -O3.

Figure 16 shows the performance of PWQ and PART relative to COPART. For PART runs with more than two threads, the additional threads were started evenly spaced along the diagonal connecting the starting and ending cells. Simplified link-state routing [5] was used to detect early termination on PART runs with more than two threads (the solution is flagged when a thread is connected to both beginning and end). PWQ performs quite poorly, but PART hits breakeven at two threads and again at five threads, achieving modest speedups beyond five threads. Theoretical energy efficiency breakeven is within the 90% confidence interval for seven and eight threads. The reasons for the peak at two threads are (1) the lower complexity of termination detection in the two-thread case and (2) the fact that there is a lower probability of the third and subsequent threads making useful forward progress: Only the first two threads are guaranteed to start on the solution line. This disappointing performance compared to results in Figure 15 is due to the less-tightly integrated hardware available in the larger and older Xeon® system running at 2.66GHz.

7 Future Directions and Conclusions

Much future work remains. First, this paper applied only one technique used by human maze solvers. Others include following walls to exclude portions of the maze and choosing internal starting points based on the locations of previously traversed paths. Second, different choices of starting and ending points might favor different algorithms. Third, although placement of the PART algorithm's first two threads is straightforward, there are any number of placement schemes for the remaining threads. Optimal placement might well depend on the starting and ending points. Fourth, study of unsolvable mazes and cyclic mazes is likely to produce interesting results. Fifth, the lightweight C++11 atomic operations might improve performance. Finally, for mazes, humiliating parallelism indicated a more-efficient sequential implementation using coroutines. Do humiliatingly parallel algorithms always lead to more-efficient sequential implementations, or are there inherently humiliatingly parallel algorithms for which coroutine context-switch overhead overwhelms the speedups?

This paper demonstrated and analyzed parallelization of maze-solution algorithms. A conventional work-queue-based algorithm did well only when compiler optimizations were disabled, suggesting that some prior results obtained using high-level/overhead languages will be invalidated by advances in optimization.

This paper gave a clear example where approaching parallelism as a first-class optimization technique rather than as a derivative of a sequential algorithm paves the way for an improved sequential algorithm. High-level design-time application of parallelism is likely to be a fruitful field of study. This paper took the problem of solving mazes from mildly scalable to humiliatingly parallel and back again. It is hoped that this experience will motivate work on parallelism as a first-class design-time whole-application optimization technique, rather than as a grossly suboptimal after-the-fact micro-optimization to be retrofitted into existing programs.

Acknowledgements

I owe thanks to Josh Triplett, Jonathan Walpole, and Cheedoong Drung for many interesting discussions, and to the anonymous referees for many helpful suggestions. I am indebted to Jim Wasko for his support of this effort.

Legal Statement

Open-source code available at `git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git`.

References

[1] ADAMATZKY, A. Slime mould solves maze in one pass . . . assisted by gradient of chemo-attractants. `http://arxiv.org/abs/1108.4956`, August 2011.

[2] AMDAHL, G. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (Washington, DC, USA, 1967), IEEE Computer Society, pp. 483–485.

[3] BECK, B., AND KASTEN, B. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 255–275.

[4] BECKER, P. Working draft, standard for programming language C++. Available: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf` [Viewed: April 3, 2011], February 2011.

[5] BERTSEKAS, D., AND GALLAGER, R. *Data Networks*. Prentice-Hall, Inc., 1987.

[6] ERICSON, C. Aiding pathfinding with cellular automata. `http://realtimecollisiondetection.net/blog/?p=57`, June 2008.

[7] ETH ZURICH. Parallel solver for a perfect maze. `http://nativesystems.inf.ethz.ch/pub/Main/WebHomeLecturesParallelProgrammingExercises/pp2011hw04.pdf`, March 2011.

[8] FOSNER, R. Scalable multithreaded programming with tasks. *MSDN Magazine 2010*, 11 (November 2010), 60–69. `http://msdn.microsoft.com/en-us/magazine/gg309176.aspx`.

[9] INMAN, J. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 277–298.

[10] KFC. Memristor processor solves mazes. `http://www.technologyreview.com/blog/arxiv/26467/`, March 2011.

[11] MCKENNEY, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html` [Viewed March 28, 2010].

[12] MUDGE, T. POWER: A first-class architectural design constraint. *IEEE Computer 33*, 4 (April 2000), 52–58.

[13] PATTERSON, D. The trouble with multicore. *IEEE Spectrum 2010* (July 2010), 28–32, 52–53.

[14] UNIVERSITY OF MARYLAND. Parallel maze solving. `http://www.cs.umd.edu/class/fall2010/cmsc433/p3/`, November 2010.

[15] WIKIPEDIA. Labyrinth. `http://en.wikipedia.org/wiki/Labyrinth`, January 2012.