Games of No Chance 5
MSRI Publications
Volume **70**, 2017

Searching for periodicity in Officers

J.P. GROSSMAN

Officers is a take-and-break game in which a move consists of removing a bean from a heap and leaving the remaining beans from that heap in exactly one or two nonempty heaps. It is an open question as to whether or not the Grundy values of this game are eventually periodic; answering this question in the positive for take-and-break games generally requires computing enough Grundy values to explicitly find the period. We contribute to this search by presenting two novel parallelization strategies that, combined with additional optimizations, accelerate the computation of Grundy values by nearly 60 times compared to the previous state of the art. The resulting implementation computes over 5 million values per second, and has computed a total of more than 140 trillion values over the course of 18 months. To date, no period has been found.

1. Introduction

Officers is an impartial game played with beans arranged into heaps. On each move, a player selects a heap with at least two beans and removes exactly one bean from the heap. If at least two beans remain in the heap, then the player may also optionally split the remaining heap into two. Officers is an example of an *octal game* [Berlekamp, Conway and Guy 2001]: a take-and-break game where each legal move can be described as removing k beans from a heap, then possibly splitting any remaining beans, leaving behind exactly 0, 1 or 2 nonempty heaps. The name "octal game" comes from the fact that such games can conveniently be described as a string of octal digits (traditionally preceded by a decimal point with trailing zeros omitted), where the k-th digit after the decimal point encodes the legal moves involving the removal of k beans from a heap. The binary representation of a digit has three bits, with bit m indicating whether or not it is legal to leave exactly m nonempty heaps. Thus, if the k-th digit is zero then it is never legal to remove exactly k beans. Under this encoding, Officers is the game .6: only the first digit after the decimal point is nonzero (only one bean can be

MSC2010: 91A46.
Keywords: Officers, octal games, periodic, Grundy values.

Take… (# beans)	1			2				
Leave… (# heaps)	**2**	**1**	**0**	**2**	**1**	**0**	***G(n)***	**Period**
.1			✓				$01\overline{0}$	1
.2 (She-loves-me-she-loves-me-not)		✓					$0\overline{01}$	2
.3 (She-loves-me-she-loves-me-not)		✓	✓				$\overline{01}$	2
.4 (Dawson's chess)	✓						0001120311…	34
.5 (She-loves-me-she-loves-me-not)	✓		✓				$\overline{01}$	2
.6 (Officers)	✓	✓					0012012312…	**???**
.7 (She-loves-me-she-loves-me-not)	✓	✓	✓				$\overline{01}$	2
.15 (Guiles)			✓	✓		✓	$\overline{1101122122}$	10
.16			✓	✓	✓		1001221401…	149459

Table 1. Simple examples of octal games, along with their periods (where known). The period of .16 was established in [Gangolli and Plambeck 1989]. For each game, checkmarks indicate the legal moves. Officers is the only single-digit octal game that is not known to be periodic.

removed), and this digit has bits 1 and 2 set (exactly 1 or 2 nonempty heaps can remain).

Richard Guy famously asked whether all finite octal games are eventually periodic [Guy 1996]. A finite octal game has a finite number of nonzero digits in its representation (equivalently, a finite number of legal moves), and "eventually periodic" means that the sequence $G(n)$, defined by the Grundy values of single heaps of size n, is eventually periodic. Thus far, no finite octal game has been shown to be aperiodic. Table 1 lists some simple examples of octal games along with their associated periods, if known. Very simple octal games can have surprisingly large periods; the period of the game .106 is 328,226,140,474 [Flammenkamp 2012]! Notably, and somewhat frustratingly, Officers is the only single-digit octal game for which Guy's question remains unanswered.

Determining the period of an octal game (if it exists) typically requires computing enough Grundy values to explicitly observe the period. Given the enormous number of values that may be needed, speed is paramount. Accelerating the computation is nontrivial because it is inherently sequential: the value $G(n)$ generally cannot be determined until $G(n-1)$ is known. Our main contribution is to show how the computation can be parallelized, leveraging both cluster computing and multicore architectures. With the additional use of established techniques for optimizing sequential code, we have been able to accelerate the computation of Grundy values for Officers by nearly 60 times relative to the fastest previously known algorithm.

Our work was specifically motivated by a desire to search for periodicity in Officers, and we focus on this game throughout the paper. However, the

```
// Use a byte array 'mex' to compute the mex function
initialize the mex array with zero
for k = 0 ... (n-1)/2
    mex[G[k] ^ G[n-1-k]] = 1;
G[n] = index of first zero in mex array
```

Figure 1. Brute-force computation of $G(n)$ in Officers.

techniques that we present are general, and could be applied to other unsolved octal games.

2. Computing Grundy values in Officers

We begin with a description of how Grundy values are computed for Officers, and we review the rare-value algorithm [Berlekamp, Conway and Guy 2001]. This algorithm, which can be applied to a large number of finite octal games, is the fastest known sequential technique for computing Grundy values.

2.1. ***Brute-force computation.*** For heaps with at least two beans, we can slightly restate the rules of Officers as follows: a player removes exactly one bean then splits the heap into exactly two heaps, one of which may be empty. This alternate formulation gives us a succinct recursive definition for $G(n)$:

$$\begin{aligned} G(0) &= G(1) = 0, \\ G(n) &= \operatorname*{mex}_{0 \le k \le (n-1)/2} \{G(k) \oplus G(n-1-k)\} \quad \text{if } n \ge 2. \end{aligned} \tag{1}$$

Here $\oplus$ denotes bitwise exclusive-or and mex is the "minimal excluded" function evaluating to the smallest nonnegative integer that does not appear in the specified set. Figure 1 lists equivalent pseudocode for computing $G(n)$. The code uses a "mex array" to mark all legal moves from a heap of size n, then searches for the first unmarked index in the array. This code is both simple and slow: since $(n-1)/2$ loop iterations are needed to compute $G(n)$, computing all Grundy values up to $G(n)$ requires $O(n^2)$ operations.

2.2. ***Rare values.*** When many Grundy values are computed for Officers, an interesting pattern emerges: when the values $G(n)$ are expressed in binary, very few of these values have an even number of bits set in positions 1, 2, 3, 5, 6, 7 and 8. Grundy values of this form are the *rare values*. Most of the finite octal games that have been studied have a similar property: for some set of bit positions (that varies depending on the specific game), few Grundy values have an even number of bits set in those positions. Just how rare are these rare values in Officers? Over 140 trillion Grundy values have been computed but only 1584 rare values have been discovered; the last known rare value is $G(20627) = 277 = 100010101_2$.

While the specific set of bits defining rare values is somewhat mysterious, some intuition can be gained as to how the set of rare values, once established, remains rare as additional Grundy values are computed. We refer to nonrare values as *common values*, i.e., values with an odd number of bits set in positions 1, 2, 3, 5, 6, 7 and 8. Consider what happens when two values are exclusive-ored. From a simple parity argument, we have:

$$\begin{aligned} \mathit{common} \oplus \mathit{common} &= \mathit{rare} \\ \mathit{rare} \oplus \mathit{common} &= \mathit{common} \\ \mathit{rare} \oplus \mathit{rare} &= \mathit{rare} \end{aligned} \tag{2}$$

Since most of the Grundy values are common, it follows that most of the elements of the set in (1) are rare, so that the first excluded integer, which is $G(n)$, is very likely to be common.

2.3. *Rare-value algorithm.* Suppose an oracle informs us that we have discovered all the rare values in Officers. In that case, to compute subsequent values of $G(n)$ we only need to consider the common values within the mex array. As such, the loop used to mark the mex array only needs to iterate over the expressions $G(k) \oplus G(n-k-1)$ where $G(k)$ is rare, so instead of $\lfloor (n-1)/2 \rfloor$ loop iterations only a fixed number (1584) are required. This means that computing all Grundy values up to $G(n)$ only requires $O(n)$ operations instead of $O(n^2)$. It also implies that the game is eventually periodic, as $G(n)$ only depends on a finite number of previous values. $G(n)$ would be trivially bounded above by 2^{12} since at most 1584 common values are marked in the mex array, so at some point some sequence of 20628 values must be repeated, after which $G(n)$ is periodic. This gives us a fairly poor bound on the period; all we can say is that it must be less than $2^{12 \times 20628}$.

In the absence of an oracle we can't assume that there are no more rare values, however we can still exploit the rare value phenomenon to accelerate the computation. We split the loop used to mark the mex array into two parts:

(1) **Mark common values.** We first iterate over $G(k) \oplus G(n-k-1)$ where $G(k)$ is rare, marking common values in the mex array. We then find the first unmarked common value, which becomes our candidate value for $G(n)$. In all likelihood this is the actual value of $G(n)$, since otherwise $G(n)$ is a rare value. To prove this, we need to be able to mark all rare values smaller than the candidate value.

(2) **Mark rare values.** Next we perform the full iteration shown in Figure 1, keeping track of the number of unmarked rare values smaller than the candidate value. As soon as this number reaches zero we have proven that

```
// 1. Find minimum excluded common value "cval"
foreach k such that G[k] is rare
    mex[G[k] ^ G[n-1-k]] = 1;
cval = first unmarked common value in mex array

// 2. Mark rare values less than cval
numr = number of unmarked rare values less than cval
for k = 0 ... (n-1)/2
    val = G[k] ^ G[n-1-k];
    if val < cval and !mex[val]
        mex[val] = 1;
        numr = numr - 1
        break if numr == 0

// 3. G[n] is either cval or a rare value less than cval
if (numr == 0)
    G[n] = cval
else
    G[n] = index of first zero in mex array
```

Figure 2. Rare-value algorithm for computing $G(n)$ in Officers.

the candidate value is $G(n)$, and we can exit the loop. Alternately, if at least one rare value never gets marked then we have discovered a new rare value.

Figure 2 lists pseudocode for the rare-value algorithm. In theory, this is still an $O(n^2)$ algorithm because the number of iterations required in the second loop is only bounded by $\lfloor (n-1)/2 \rfloor$. In practice, however, less than 3000 iterations are required on average to prove that the candidate value is $G(n)$, so in effect the algorithm is only $O(n)$. On a 2.666 GHz Intel Xeon E5430,[1] the rare-value algorithm computes roughly 88,000 values per second.

3. Accelerating the computation

We now describe a number of parallelization and optimization techniques that can be used to accelerate the computation of Grundy values in Officers far beyond the performance of a baseline implementation of the rare-value algorithm.

3.1. ***Parallelization through speculation.*** The rare-value algorithm would be much faster if the second loop could be eliminated. To this end, suppose we simply assume that there are no more rare values, and speculatively compute Grundy values based on this assumption. Figure 3 lists the (greatly simplified) pseudocode for this speculative computation. However, we still need to verify the assumption which effectively requires the original computation, so at first it seems that we haven't actually gained anything.

[1]All performance measurements in this paper were performed on the same machine. Code was written in C and compiled using the Intel compiler with the -fast option.

```
foreach k such that G[k] is rare
    mex[G[k] ^ G[n-1-k]] = 1;
G[n] = first unmarked common value in mex array
```

Figure 3. Speculative computation of $G(n)$ in Officers, assuming no more rare values.

The key observation is that once a large number of speculative values have been computed, we can perform the verification in parallel on a cluster. Specifically, the sequence of values can be partitioned into intervals $[N_j, N_j+1)$, and for each interval a processor can use the rare-value algorithm to verify that the Grundy values in $[N_j, N_j+1)$ are correct assuming that all previous speculative values are correct. Once all intervals have been verified, an inductive proof is formed that the entire sequence of speculative values is correct.

With a sufficiently large cluster we can verify the speculative values as fast as they can be generated, so our overall performance is only limited by how quickly we can perform the speculative computation. A direct implementation of the pseudocode in Figure 3 computes roughly 196,000 values per second, representing a speedup of 2.2 times over the rare-value algorithm.

3.2. ***Sequential optimizations.*** Now that the bulk of the computation has been reduced to a single tight loop, we can apply two standard optimizations to obtain additional speedups. First, all known Grundy values in Officers are less than 512 and can therefore be represented using 9 bits. Moreover, since we are only computing common values, the value of bit 8 is implied by the values of the other bits, so in fact 8 bits suffices to represent the values and we can store $G(n)$ in a byte array. We never need to recover the ninth bit, and the loop used to mark the mex array is unmodified. The size of the mex array is reduced from 512 entries to 256 entries, since dropping bit 8 creates a one-to-one mapping from common values in [0, 512) to numbers in [0, 256). The only change to the computation is that we need to take this mapping into account when we search the mex array for the first unmarked common value. Additionally, we need to validate the assumption that the Grundy values remain less than 512. Using a byte array for $G(n)$ improves the performance to 321,000 values per second.

The second optimization is to fully unroll the main loop, since we know exactly how many iterations there are as well as the value of $G(k)$ in each iteration. Figure 4 lists the code for the unrolled loop. When compiled, the resulting assembly code contains three instructions per iteration. This optimization further improves performance to 850,000 values per second.

Searching the mex array for the first unmarked common value is also implemented as a loop. This search is much less expensive than the main loop,

```
mex[0 ^ G[n-1]] = 1
mex[0 ^ G[n-2]] = 1
mex[1 ^ G[n-3]] = 1
mex[0 ^ G[n-5]] = 1
mex[1 ^ G[n-6]] = 1
mex[1 ^ G[n-9]] = 1
…
G[n] = first unmarked common value in mex array
```

Figure 4. Speculatively computing $G(n)$ in Officers using an unrolled loop.

```
int compute_mex ()
    if (!mex[2])
        return 2;
    if (!mex[3])
        return 3;
    if (!mex[4])
        return 4;
    if (!mex[5])
        return 5;
    if (!mex[8])
        return 8;
…
```

Figure 5. Evaluating the mex function using an unrolled loop.

representing only 10% of the total compute time even after the main loop is unrolled, but we can still obtain some additional performance benefit from unrolling this loop as well. Figure 5 lists the code for the unrolled mex computation; the compiled code again contains three instructions per iteration. With both loops unrolled, performance increases to 912,000 values per second, more than 10 times faster than the baseline rare-value algorithm.

3.3. ***Multithreading.*** Since the value of $G(n)$ depends on $G(n-1)$, it is not immediately clear how the computation can be further parallelized. Observe, however, that most of the computation of $G(n)$ does not depend on $G(n-1)$, which is only required for a single iteration of the unrolled loop. We can therefore use two threads with a baton-passing approach: a thread marks as much of the mex array as it can without knowing $G(n-1)$, then it receives the value of $G(n-1)$ (the baton) from the other thread, then it finishes computing $G(n)$ and passes this value to its partner. Figure 6 lists pseudocode for this algorithm.

This technique naturally generalizes to m threads. Thread j is responsible for computing $G(n)$ where $n \equiv j \pmod m$, and the baton comprises the previous $m-1$ Grundy values. Figure 7 plots the performance of this multithreaded computation as the number of threads is increased from 1 to 8 ("Standard mex" curve). The performance with 8 threads is 3,672,000 values per second.

```
mex[0 ^ G[n-2]] = 1
mex[1 ^ G[n-3]] = 1
mex[0 ^ G[n-5]] = 1
…
receive G[n-1] from the other thread
mex[0 ^ G[n-1]] = 1
G[n] = first common value not in mex
pass G[n] to the other thread
```

Figure 6. Multithreading the computation of $G(n)$ using baton passing.

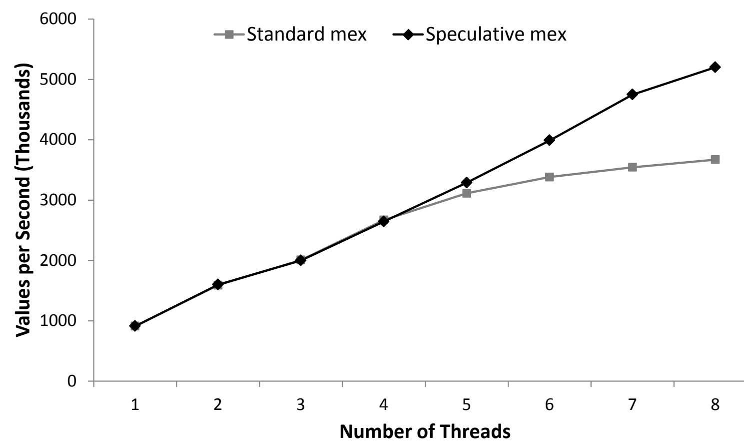

Figure 7. Performance of multithread computation.

In Figure 7 we can see that performance begins to level off beyond 4 or 5 threads. The reason for this is that each thread must wait to receive the previous Grundy values before evaluating the mex function, and must then compute the mex function before forwarding Grundy values to the next thread (Figure 8). This places an upper bound on the performance that can be achieved: the time to compute a value must be at least the time required to evaluate the mex function

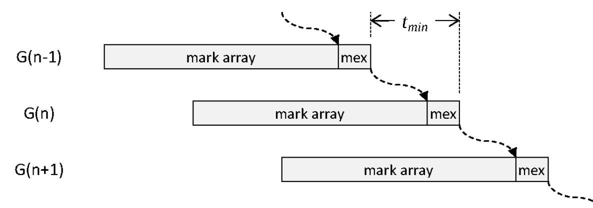

Figure 8. Standard mex computation. A thread waits for the baton, then evaluates the mex function to compute $G(n)$, then passes the baton to the next thread. Performance is limited by the time to compute the mex function plus the time to pass the baton.

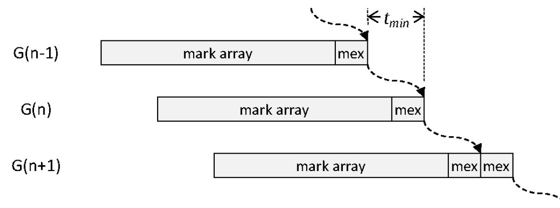

Figure 9. Speculative mex computation. A thread speculatively computes $G(n)$ before receiving the baton, then checks to see if it needs to recompute $G(n)$ after receiving the baton. In the overwhelming majority of cases the speculative computation is correct, so performance is only limited by the time to pass the baton. If the speculative computation is incorrect then the mex computation is repeated, as shown for $G(n+1)$.

plus the inter-thread communication latency. As we approach this upper bound, additional threads offer little incremental benefit.

Suppose that, instead of waiting to receive the previous $m-1$ Grundy values, each thread speculatively evaluates a candidate Grundy value $G'(n)$, even though it cannot finish marking the mex array until it receives data from the previous thread. Under what circumstances will this speculative computation be correct, so that $G(n) = G'(n)$? By definition of the mex function, the value $G'(n)$ will be unmarked, whereas all smaller values will be marked. Once the previous $m-1$ Grundy values are received, a small set of additional entries must be marked (only 5 when $m = 8$). $G'(n)$ is correct if and only if it is not in this set. This determination is computationally inexpensive, requiring only a handful of processor cycles. If it turns out that $G'(n)$ is incorrect then the mex function must be recomputed after the remaining values have been marked, but empirically we have found this case to be extremely rare.

Figure 9 illustrates the speculative mex computation, which has the effect of removing the mex evaluation from the critical path. Now, performance is only limited by the inter-thread communication latency, allowing the computation to scale to a larger number of threads. Figure 7 shows the performance of this modified algorithm ("Speculative mex" curve), which exhibits much better scaling with more than four threads. With 8 threads the performance increases to 5,201,000 values per second — 59 times faster than the rare-value algorithm.

3.4. ***Performance summary.*** Table 2 summarizes the performance of the algorithms described in the previous sections, showing both incremental speedup as each successive optimization is applied as well as total speedup relative to the baseline rare-value algorithm. The final speculative mex multithreaded algorithm

Algorithm	Values per millisecond	Incremental speedup	Total speedup
Rare values	88	1.0	1.0
Speculative	196	2.2	2.2
Byte array	321	1.6	3.6
Unroll loops	912	2.8	10.4
Multithreaded	3672	4.0	41.7
Speculative mex	5201	1.4	59.0

Table 2. Performance of the various algorithms presented in earlier sections, and for the first row (rare values) in [Berlekamp, Conway and Guy 2001].

takes, on average, 192 ns to compute each Grundy value, or equivalently 512 processor cycles.

4. Searching for periodicity

Ultimately, our goal is to search for periodicity within the sequence of Grundy values. To prove that the sequence is periodic with period p and preperiod d, it suffices to compute values up to $G(2p+2d)$, verifying the periodicity within this range; periodicity for all larger values then follows from a simple induction argument. But how can we efficiently determine candidate values for p and d?

Observe that, if the hypothesis that there are no more rare values beyond $G(20627)$ holds, then each Grundy value only depends on the previous 20628 values. We can therefore search for two identical sequences of 20628 consecutive values. This still sounds like an expensive proposition, but there is an efficient online algorithm for performing this search based on two further simplifications.

The first simplification is to consider the sequence $H(n)$ defined by computing a 64-bit cyclic redundancy check (CRC) of the values $G(n)$, $G(n-1)$, $\dots$, $G(n-20627)$. We then search for n_0 and n_1 such that $H(n_0) = H(n_1)$, which is much cheaper than comparing two complete sequences of 20628 values. The full CRC computation is expensive, but it does not need to be repeated for each $H(n)$: there is a fast incremental way to compute $H(n)$ from $H(n-1)$, $G(n)$ and $G(n-20628)$ that involves two shifts, two table lookups, and three exclusive-ors. Note that this algorithm may produce false positives so that we still need to do a full sequence comparison for candidate values of n_0 and n_1, but with a 64-bit CRC these false positives are extremely rare.

The second simplification is to restrict n_0 to powers of 2. For each interval $[2k, 2k+1)$ we compute $H(2k)$, then we compare $H(n)$ to $H(2k)$ within the interval. In so doing we almost definitely won't find the smallest values of n_0 and n_1 such that $H(n_0) = H(n_1)$, but we are at least guaranteed to compute

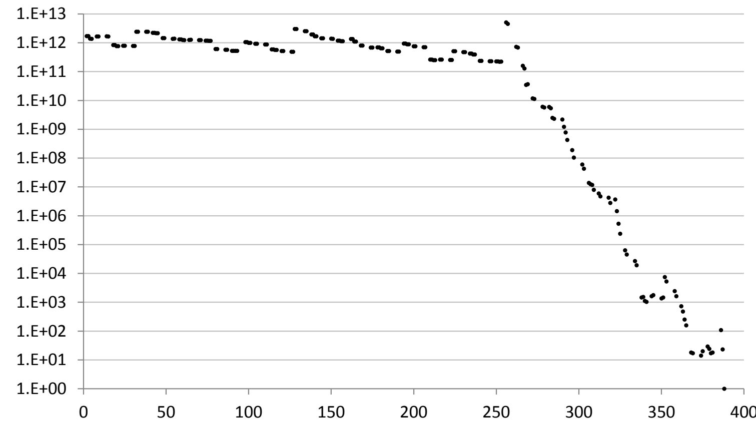

Figure 10. Histogram of common values within the first 140 trillion Grundy values of Officers.

fewer than twice as many values as we need to since there is always a power of 2 between n_0 and $2n_0$.

Augmenting the computation to simultaneously search for periodicity by looking for a repeated CRC value has a small (less than 4%) performance impact, and we are able to process over 5 million Grundy values per second. To date, we have computed over 140 trillion Grundy values without a single candidate CRC match. The largest Grundy value we have observed is 388, whose first (and thus far only) appearance is at $G(7{,}014{,}808{,}364{,}046)$. The last "new" Grundy value we have observed is 380, whose first appearance is at $G(23{,}209{,}561{,}059{,}317)$. Figure 10 shows a histogram of the common Grundy values. A certain amount of structure is evident in the histogram for the values less than 256; the pattern for the remaining values is less clear.

5. Discussion

The hardware used for this work offered a maximum of 8 processor cores, but the performance graph of the speculative mex algorithm in Figure 7 strongly indicates that there is additional performance to be gained by increasing the number of threads. The current trend in commodity processor technology is to provide an increasing number of processor cores per die with little or no increase in clock speed; Intel recently announced their Knight's Landing architecture with over 60 cores [Intel 2014]. This being so, future general-purpose architectures will likely provide additional multithreading speedups for the algorithm described in this paper. It may also be possible to tailor the algorithm for implementation on a graphics processing unit (GPU); recently GPUs have become popular for

high-performance computing due to their ability to support a large number of threads performing identical computations.

An alternate approach to improving performance would be to implement the unrolled single-threaded algorithm directly in hardware using a field programmable gate array (FPGA). The algorithm is extremely well suited to such an implementation, and would easily be able to generate a new value on every clock cycle. A modern FPGA can be clocked at around 500 MHz, which would thus generate 500,000,000 values per second — well over 5000 times faster than the baseline rare-value algorithm. At this speed the problem of validating the speculative values becomes quite challenging: a cluster with nearly 6000 processing cores would be required to keep pace with the data produced by a single FPGA!

It is entirely possible that Officers is eventually periodic, but with a period or preperiod so astronomically large that no direct linear search could find the period. If so, all hope is not necessarily lost; analysis of the data produced by the current brute-force search may reveal patterns giving rise to more advanced algorithms that can produce N values per step for some large or increasing N.

The algorithms in this paper are general and could be applied to any octal game that has a set of rare values, as well as certain other take-and-break games such as Grundy's game. Officers happens to be particularly well suited to fast computation on commodity processors for two reasons. First, all known values are less than 512, allowing the computation to be performed using bytes. Second, and more importantly, all the known rare values occur within the first 20628 Grundy values, which means that only the most recent 20628 Grundy values need to be kept in memory for the unrolled speculative computation described in Section 3.1. With such a small memory footprint, the computation data easily fits within the processor cache. Other octal games may not share these properties, but would still benefit from the parallelization techniques that we have presented.

Acknowledgements

The computations described in this paper were performed on the Department of Mathematics and Statistics Compute Cluster at Dalhousie University. We would like to thank Balagopal Pillai for his assistance in using and managing the cluster resources.

References

[Berlekamp, Conway and Guy 2001] E. Berlekamp, J. H. Conway, and R. K. Guy, *Winning ways for your mathematical plays*, vol. 1, A K Peters, Wellesley, MA, 2001.

[Flammenkamp 2012] A. Flammenkamp, "Sprague–Grundy values of octal games", web page, 2012, http://wwwhomes.uni-bielefeld.de/achim/octal.html. Retrieved September 28, 2014.

[Gangolli and Plambeck 1989] A. Gangolli and T. Plambeck, "A note on periodicity in some octal games", *Internat. J. Game Theory* **18**:3 (1989), 311–320.

[Guy 1996] R. K. Guy, "Unsolved problems in combinatorial games", pp. 475–491 in *Games of no chance* (Berkeley, 1994), Math. Sci. Res. Inst. Publ. **29**, Cambridge Univ. Press, 1996.

[Intel 2014] "Intel re-architects the fundamental building block for high-performance computing", press release, Intel Corporation, 2014, http://tinyurl.com/intel-re.

jpg@alum.mit.edu *D. E. Shaw Research, 120 West 45th Street, 39th Floor, New York, New York 10036, United States*