# The p196_mpi implementation of the reverse-and-add algorithm for the palindrome quest.

Romain Dolbeau

March 24, 2014

## 1 Introduction

To quote John Walker, the first person to brute-force the problem [1]:

> Pick a number. Reverse its digits and add the resulting number to the original number. If the result isn't a palindrome, repeat the process. Do all numbers in base 10 eventually become palindromes through this process? Nobody knows.

After the three years of computing that gave its title to Walker paper, many others became interested in the issue. The problem itself is older, and was already mentioned in David Wells book on the subject of interesting numbers [2]. Many people worked on the quest over the decades. The numbers whose existence is questioned were christened Lychrel by Wade VanLandingham, who maintains a comprehensive web site dedicated to them [3]. Some scholarly works have been published on theoretical rather than computational aspects [4].

This paper describes the implementation of the `p196_mpi` code, a distributed code dedicated to compute iterations of the palindrome quest as fast as possible. As of the end of 2013, this is the fastest known code to compute the reverse-and-add algorithm in base 10. It was the first code to break the 300 million digits mark and subsequently to reach 600 million digits.

No such computational effort will positively answer the question "is 196 a Lychrel number?". It can only negatively answer, by finding a palindrome, something that is highly unlikely.

## 2 Basic Algorithm

The basic algorithm can be expressed extremely simply, as it is merely a sum. If storing the data one digit at a time in an array, then the basic algorithm for one iteration looks like this:

```
char carry = 0;
for (i = 0 ; i < numdigits ; i++) {
  char intermediate = current[i] +
                      current[numdigits - (i + 1)] + carry;
  carry = (intermediate >= base) ? 1 : 0;
  next[i] = carry ? intermediate - base : intermediate;
}
if (carry) {
  next[numdigits] = 1;
  numdigits++;
}
```

Where `current` holds the current value, `next` holds the results of the iteration, and `numdigits` is the current number of digits in the value. This is basically a by-the-book sum. The only specificity is the operands: the reverse-and-add process means there is just one (the `current` array), read both forward and backward. Of course, the carry propagation means the algorithm is intrinsically sequential.

## 3 Vector algorithms

There is at least two different ways to "vectorize" (i.e. exploit SIMD parallelism) the algorithm. The first is to separate the actual computations (the additions) from the carry propagation, and iterate the carry propagation until no more carry propagation are required. The second is to transform the digit-level carry propagation into a block-level carry propagation. The actual code uses an hybrid of both solutions.

### 3.1 Splitted computation-propagation algorithm

The splitted computation-propagation algorithm can be made to work on almost any vector-capable hardware. However, as it is memory bandwidth-consuming, it wasn't implemented by itself in `p196_mpi`, only in prototype code, such as for GPU computing. It is instead used implicitly in the distributed aspect of the code in its "block" version.

The algorithm is very simple: an additional array of carries is introduced, along with a keep-going variable and an iteration number variable. The first iteration computes all the sums of digits, but stores the carries into the new array instead of propagating them. Then, as long as there is a least a non-zero carry (indicated by the `keepgoing` variable), carries are added to the proper digits, and a new set of carries is computed. The maximum number of iterations is the maximum number of digits through which a carry is effectively propagated. In practice, it is very seldom more than a dozen iterations [1].

[1]After the main additions are done, a carry will only propagate through a consecutive sequence of 9

```
iteration = 1;
for (i = 0 ; i < numdigits ; i++)
  carries[i] = 0;
for (i = 0 ; i < numdigits ; i++) {
  char localcarry = 0;
  char intermediate = current[i] +
                current[numdigits - (i + 1)];
  localcarry = (intermediate >= base) ? 1 : 0;
  next[i] = localcarry ? intermediate - base : intermediate;
  carries[i] = localcarry;
  if (localcarry)
    keepgoing = true;
}
while (keepgoing) {
  keepgoing = false;
  for (i = 0 ; i < numdigits ; i++) {
    char localcarry = (i >= iteration ? carries[i - iteration] : 0);
    char intermediate = next[i] + localcarry;
    localcarry = (intermediate >= base) ? 1 : 0;
    next[i] = localcarry ? intermediate - base : intermediate;
    carries[i] = localcarry;
    if (localcarry)
      keepgoing = true;
  }
  iteration ++;
}
if (carries[numdigits-1]) {
  next[numdigits] = 1;
  numdigits++;
}
```

All loops in the code are parallel; the only issue is when writing to the `keepgoing` variable, as it is required that if multiple simultaneous writes occur, at least one will succeed (they all write the same value, so which one is not relevant to the algorithm). In case the parallel system doesn't guarantee this, then an atomic update or a lock is required.

The implementation shown above uses a carry array as big as the digit array. There is nothing to prevent the algorithm to use a smaller carry array, by considering sub-array of digits instead of a single digit. For instance, if a carry is sequentially propagated through a sub-array of four digits, then the carry array needs only be one-fourth the size of the digit array. The number of iterations required is only the maximum number of sub-arrays that a carry will propagate through, which for larger-than-four sub-arrays is usually at most four iterations.

## 3.2 Block-level carry propagation algorithm

The block-level carry propagation algorithm is based on the availability of multi-byte additions. Those are usually limited to 8 bytes, a value still extremely useful. The principle of the algorithm is to “offset” all digits but the most significant one in a block, so that a conventional multi-byte addition will implicitly propagate the carry through the block. When storing a single digit in each byte, the magic number is 246 (F6 in hexadecimal), i.e. offsetting the digits so that 9 becomes 255, while 10 becomes 0 with propagation of a carry to the next byte. The following is an example exploiting a 32 bits addition. 4 digits have been stored in `forward`, while the 4 digits from the reversed number are stored in `backward`.

```
unsigned int offseted = forward + 0x00F6F6F6;
unsigned int result = offseted + backward;
if ((result & 0x000000FF) >= 0x000000F6)
  result -= 0x000000F6;
if ((result & 0x0000FF00) >= 0x0000F600)
  result -= 0x0000F600;
if ((result & 0x00FF0000) >= 0x00F60000)
  result -= 0x00F60000;
carry = (result & 0xFF000000) >= 0x0a000000 ? 1 : 0;
result = carry ? result - 0x0a000000 : result;
```

It may seems quite a lot of instructions for only 4 bytes. And some of them are very annoying control flow instructions. But the conditionals can be replaced by combinations of boolean and arithmetic operations, and in practice 64 bits additions are used. And the main advantage is a much better exploitation of cache & memory bandwidth, and register storage.

## 3.3 SSE4 implementation

The SSE4 implementation is essentially a dual-block, unrolled version of the block-level carry propagation algorithm. 64 bytes (512 bits, 64 digits) are loaded for both the number and its reverse. In the case of the reverse, unaligned memory access and the `pshufb` [5] instructions greatly simplifies the implementation in SSE4. Each of four XMM registers holds two blocks of 8 digits.

Vector 8 bits additions introduce the magic number 246, then vector 64 bits additions do the arithmetic with the implicit carry propagation inside each block. 246 must then be subtracted for all digits that didn’t create a carry; this is implemented by exploiting the fact that they have their most significant digit sets, and are therefore negative in signed 2-complement arithmetic. This helps create a mask to subtract 246 only where that is required.

There is eight different blocks of 8 digits. The carries from the seven less significant blocks are propagated iteratively be repeating the previous operations after extracting an reinserting the carries. When this is done, the resulting 64 digits are stored using streaming stores. The last carry itself is kept as input for the next block.

Finally, some prefetching was introduced to better exploit the available memory bandwidth.

## 4 Distributed algorithm & implementation

The distributed algorithm is essentially the splitted computation-propagation algorithm using very large sub-arrays. The entire array of digits is split in many sub-arrays, one per process. All sub-arrays are computed in parallel, each using the SSE4 implementation described above (looping over all the 64 bytes blocks in the sub-array). The carry are virtually "stored" by being sent to the next sub-array, then propagated; this is repeated until there is no more non-zero carry to propagate (propagating through more than one large sub-array is exceedingly rare in practice).

In the actual implementation, the whole array is splitted into two sub-arrays per process. The first process owns the first and last sub-arrays, the second process owns the second and next-to-last sub-array, and so on, until the last process owns the two middle sub-arrays. The reason is data availability: computing the first digit of the first sub-array requires the last digit of the last sub-array, from the "reverse" part of the "reverse-and-add" process. Storing them in the same process saves a large amount of data exchange between processes. However, the sub-arrays are almost never perfectly aligned. First, the number of digits is unlikely to be an integer multiple of the number of processes. Second, there is a need for expansion, are each carry propagation beyond the most significant digit increases the digit count by one. So the last sub-array is smaller than the others for expansion.

The sequence of operations illustrated in figure 1 is therefore:

1. Check whether we already have a palindrome and should stop (step 1);
2. Do the additions & carry propagation inside both sub-arrays (step 2);
3. Send carries for both sub-arrays to neighboring processes, propagate. Iterate if needed (step 3);
4. Process 0 lets everyone know whether there is an extra digit (step 4);
5. Send new values that are needed for the next iteration to the neighboring processes (step 5).

The last step is required because of the imperfect matching of sub-arrays inside each process. This is a critical part: if the last sub-array is much smaller than the others, then a lot of data exchange is required. But the last sub-array grows, and if it is close in size to the others, it will quickly catch-up in size. Whenever the last sub-array becomes as large as the other, a redistribution of data is needed (step 6 in figure 1), growing all sub-arrays evenly at the expense of the last sub-array. After each such redistribution, the size discrepancy is

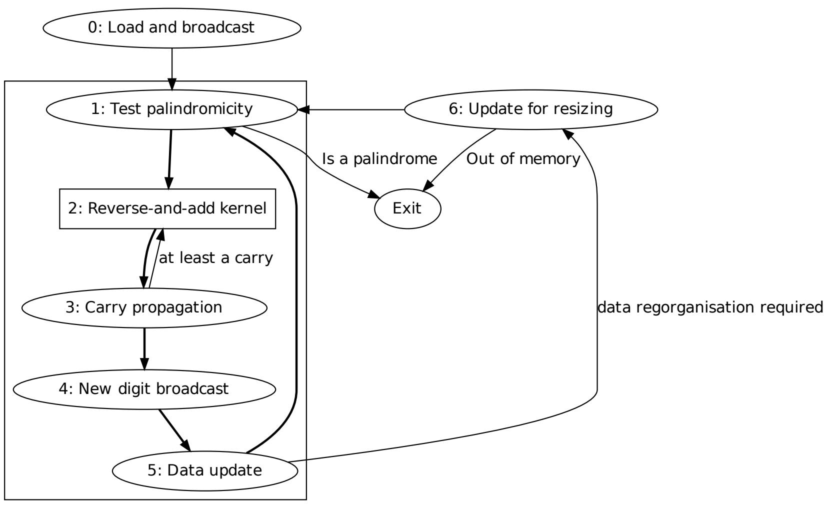

Figure 1: Overview of the distributed algorithm

maximal, and therefore the cost of the data propagation (step 5) is maximal. This cost goes down as the last sub-array grows, and it is minimal just before a new redistribution is required.

## 5 Other code

In addition to the actual reverse-and-add process, the code needs to check after each iteration if the result is a palindrome. The cost lies mostly in the required global reduction operation to combine the answer, as the comparison of digits stops as soon as a discrepancy is found, usually very quickly.

The rest of the code involves mostly the reading and dumping of data and other housekeeping functions.

## 6 Performance considerations

There is three main contributors to the execution time when running under normal condition (i.e., with a sufficiently large problem for the given number of cores).

1. The actual computations, mostly during the initial sum;
2. The carry propagation among processes;
3. The data exchange of new values.

The actual computations, using a properly written kernel, are essentially memory-bound. The `current` array must be read almost twice (once both way), with little reuse except in the middle when the direct and reverse accesses overlap. The `next` array is written once with no reuse. Observed performance under ideal condition is coherent with this view, with a number of digits-per-second (the standard metric for the problem) at about one third of the measured STREAM triad bandwidth. Note that the number of digits-per-second measure exactly what it says, i.e. summing a one million digits number with its own reverse counts for one million digits. If that work is done in a second, then the speed is one million digits-per-second (d/s).

Carry propagation is purely latency bound (only one byte is sent by one process to each neighbor), while the cost of data exchange is highly volatile as explained in section 4.

An interesting comparison can be made between the documented work of John Walker [1] and the currently available systems. As the name implies, it took John Walker three years to reach a million digits. This corresponds to a total work of 1208405465053[2] digits summed. This can be easily counted by redoing the entire work (a matter of minutes at most on any modern computer). This can also be approximated by the sum of all numbers of digits up to the chosen size, multiplied by the average number of times each size occurred, i.e. if it took $M$ iterations to reach $N$ digits: $((N \times (N+1))/2) * (M/N)$ or $(((1000000 \times (1000000+1))/2) * (2415836/1000000)) = 1207919207918$, with an error of less than 0.5%. If we take the almost three years at face value, that's an average speed of about 13000 d/s.

The code described in this paper was run on large clusters, occasionally with a hundred nodes or more. In one particular case, each "Nehalem"-based node had an available bandwidth of about 40 to 45 GB/s and a hundred nodes were used. Despite overheads from the MPI communications, the code was able to reach an average speed of over $1.1 \times 10^{12}$ d/s. Over one second periods, peaks were in excess of $1.25 \times 10^{12}$ d/s.

Although the parallelism is much more abundant for numbers with a larger numbers of digits and therefore not quite directly comparable, this means that given enough hardware, the code can in less than one second perform more work than the entire original three years of computing represented.

## 7 Conclusion & alternatives

As of late 2013, this is the fastest known implementation of the base-10 reverse-and-add algorithm. At least two improvements could theoretically be made to the code.

First, during the actual reverse-and-add computations, the data could be used to compute both sub-arrays simultaneously. This would theoretically save on memory bandwidth, a major bottleneck of the current implementation.

[2] $1.208405465053 \times 10^{12}$

Second, each digit is currently stored in a full byte. If each digit was stored in a nybble (half a byte), then the memory bandwidth requirement would be cut in half (along with the memory size itself). However, this would break the block-level carry propagation algorithm, as it effectively requires the ability to store up to 20 different values in the storage space of a single digit. So a different algorithm, still mostly memory-bound, would be required to effectively use this alternate data storage format.

Finally, no matter how fast the reverse-and-add process, such code will never prove the existence of Lychrel numbers. While it is extremely unlikely that a palindrome will be found, not finding one does not mean there isn't one. Only a theoretical demonstration would work, as exists for some other bases [3].

## References

[1] J. Walker. (May 25th, 1990) Three years of computing. [Online]. Available: http://www.fourmilab.ch/documents/threeyears/threeyears.html

[2] D. Wells, The Penguin Dictionary of Curious and Interesting Numbers. Penguin Books, 1986, no. 0-14-008029-5.

[3] W. VanLandingham. 196 and other Lychrel numbers. [Online]. Available: http://www.p196.org

[4] V. Prosper and S. Veigneau, "On the palindromic reversal process," CALCOLO, vol. 38, no. 3, pp. 129–140, 2001. [Online]. Available: http://dx.doi.org/10.1007/s100920170002

[5] Intel Corporation, Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, March 2013, no. 325383-046.

[3] For instance see http://www.mathpages.com/home/kmath004/kmath004.htm and http://www.mathpages.com/home/dseal.HTM